

§ 3.3.2 输出语句(PRINT 语句和 WRITE 语句) .....	35
§ 3.4 格式语句.....	36
<b>第四章 CM Fortran 控制结构.....</b>	<b>38</b>
§ 4.1 条件结构.....	38
§ 4.2 CASE 结构 .....	45
§ 4.2.1 CASE 结构的一般形式 .....	45
§ 4.2.2 CASE 结构的控制执行.....	46
§ 4.2.3 CASE 结构的标识符 .....	47
§ 4.3 循环结构.....	48
§ 4.3.1 DO 循环结构 .....	48
§ 4.3.2 DO TIMES 循环结构 .....	50
§ 4.3.3 DO WHILE 循环结构 .....	51
§ 4.3.4 EXIT 语句与 CYCLE 语句 .....	52
§ 4.3.5 循环结构的嵌套.....	54
§ 4.3.6 隐含 DO 循环.....	57
<b>第五章 CM Fortran 数组与数据处理.....</b>	<b>59</b>
§ 5.1 数组的定义和有关说明.....	59
§ 5.1.1 数组的定义和数组说明符.....	59
§ 5.1.2 数组说明语句.....	60
§ 5.1.3 数组的下标与存储次序.....	62
§ 5.1.4 数组段(部分数组).....	64
§ 5.2 数组的赋值、运算和输入/输出.....	67
§ 5.3 不同形式的数组说明.....	72
§ 5.4 数组的屏蔽.....	76
§ 5.5 数组元素的分配语句 FORALL .....	81
§ 5.6 动态分配.....	83
<b>第六章 CM Fortran 数组变换.....</b>	<b>88</b>
§ 6.1 数据移动函数.....	88
§ 6.1.1 循环移动函数 CSHIFT .....	88
§ 6.1.2 截止移位 EOSHIFT 函数 .....	90
§ 6.1.3 矩阵的转置函数.....	92
§ 6.2 数组的归约函数.....	93
§ 6.2.1 求数组中最大元素的值函数.....	94
§ 6.2.2 数组的乘积.....	95
§ 6.2.3 求和函数.....	96
§ 6.2.4 计数函数.....	97

---

§ 6.2.5 ALL 和 ANY 函数 .....	97
§ 6.3 数组的构造函数 .....	99
§ 6.3.1 对角线构造数组函数 .....	99
§ 6.3.2 数组归并构造函数 .....	100
§ 6.3.3 数组的压缩与扩展函数 .....	100
§ 6.3.4 复制函数和扩展函数 .....	102
§ 6.3.5 重新整形函数 .....	104
§ 6.4 向量点积和矩阵的乘法 .....	106
§ 6.4.1 向量点积 DOTPRODUCT .....	106
§ 6.4.2 矩阵的乘法 MATMUL .....	107
§ 6.5 数组应用实例 .....	108

### 第三部分 并行程序通信

<b>第七章 CMMD 概述 .....</b>	<b>119</b>
§ 7.1 程序模型 .....	119
§ 7.2 通信协议 .....	119
§ 7.3 CMMD 的输入输出 .....	121
§ 7.4 CM-5 的体系结构 .....	121
§ 7.5 一个简单的CMMD 程序 .....	122
<b>第八章 CMMD 同步通信函数 .....</b>	<b>124</b>
§ 8.1 缓冲区和数组 .....	124
§ 8.2 发送消息函数 .....	124
§ 8.3 接收信息函数 .....	126
§ 8.4 同时发送和接收函数 .....	127
§ 8.5 两个节点之间的信息交换函数 .....	128
§ 8.6 节点信息函数(辅助函数) .....	129
§ 8.7 信息检测函数 .....	130
§ 8.8 信息存取器函数 .....	130
<b>第九章 CMMD 异步通信函数 .....</b>	<b>132</b>
§ 9.1 异步发送函数 .....	132
§ 9.2 异步接收函数 .....	133
§ 9.3 非块化发送函数 .....	134
§ 9.4 异步检测函数 .....	135
§ 9.5 MCR 存取器函数 .....	135
§ 9.6 释放信息控制块子程序 .....	136
§ 9.7 等待异步信息子程序 .....	136
§ 9.8 节点广播函数 .....	137

---

<b>第十章 CMMD 应用实例</b>	138
§ 10.1 例题及其算法	138
§ 10.2 程序及其说明	140
附录 10.1 CM Fortran 源程序	141
附录 10.2 CM Fortran 编程数值结果的图形显示	146
附录 10.3 安装在国立澳大利亚大学的 CM-5 系统	148

#### 第四部分 并行计算机编程环境与分布式程序设计

<b>第十一章 PVM</b>	151
§ 11.1 PVM 概述	151
§ 11.2 启动与配置 PVM	152
§ 11.3 编写 PVM 应用程序	155
§ 11.3.1 C 语言编程示例	155
§ 11.3.2 Fortran 语言编程示例	159
§ 11.3.3 编写应用程序应该注意的几个问题	162
§ 11.3.4 编译和运行 PVM 应用程序	165
§ 11.3.5 程序调试	165
§ 11.4 PVM 库函数使用指南	167
§ 11.4.1 进程控制类函数	167
§ 11.4.2 信息类函数	170
§ 11.4.3 动态配置类函数	174
§ 11.4.4 信号函数	175
§ 11.4.5 错误信息处理函数	177
§ 11.4.6 信息传递类函数	177
§ 11.4.7 动态进程组织类函数	189
§ 11.5 PVM 应用实例	193
<b>第十二章 Linda</b>	199
§ 12.1 C-Linda	199
§ 12.1.1 Tuple 空间的数据结构	199
§ 12.1.2 C-Linda 对 Tuple 空间的存取操作	200
§ 12.1.3 Tuple 配置规则	201
§ 12.1.4 C-Linda 的程序结构、编译、运行	203
§ 12.2 C-Linda 应用实例	205
附录 12.1 串行程序	207
附录 12.2 同步并行程序	209
附录 12.3 异步并行程序	213
<b>参考文献</b>	218

# 第一部分 并行计算机和并行算法

第一章 并行计算机概述

第二章 并行算法概述

## 概要

本部分介绍作为并行程序设计基础的并行计算机和并行算法概况。通过这部分的介绍，读者可了解到并行计算机和并行算法发展动向，以及编程的 SIMD 数据并行模式和 MIMD 信息传递模式等，认识并行设计的必要性。



# 第一章 并行计算机概述

所谓并行处理是指计算机同时处理多条指令、多个数据或多个任务。要实现对一个问题的并行处理光有可并行处理的硬件结构是不够的，还必须有支持并行处理的软件和处理问题的并行算法。现代并行处理技术包含并行的系统结构、支持并行处理的软件和实现并行处理的算法等方面的内容。本章仅就并行处理的系统结构作一概要的叙述。

## § 1.1 并行处理系统

并行处理系统是由  $n$  台处理机或  $n$  台等效的处理机(器)组成，受统一的操作系统控制，解算同一问题，系统硬件的价格不能大于单处理机(器)硬件价格的  $n$  倍，系统的运算速度要能接近单处理机(器)速度的  $n$  倍。并行处理系统是提高处理速度和解决大规模问题的重要因素，展现出多处理机组成更大的计算机系统以及由超大规模集成电路(VLSI)组成计算机的前景，近二十年对并行处理技术的研究一直是计算机科学领域内的重大课题之一。并行处理系统是一门综合性的学科，它涉及并行的系统结构、并行算法、并行程序语言和并行操作系统等。下面仅对并行计算机发展、分类、语言等作一粗略描述，以便我们对并行处理系统有一个初步的了解。

### § 1.1.1 从串行机到并行机

众所周知，第一代计算机全部结构是按串行方式工作的，它遵循 Burks 等人于 1946 年设计的存储程序计算机的基本概念，通常称为 Von Neumann 结构。串行计算机的主要特点是计算机每一操作必须按顺序执行，或一次执行一个操作（如存储器的取数或存数，运算或逻辑操作，输入或输出操作）。这些早期计算机能存储并完成单位计算，应用于位串行。

1953 年 IBM 公司完成了具有并行运算的第一台商业计算机——IBM701 机，从而拉开了并行计算机的序幕。在并行计算机的发展过程中，经历了两个水平层次，这两个层次都要试图增强计算特殊类别的操作运算，早期努力的中心点是关于发展高速运行的巨型计算机，以解决大量科学上的问题，例如精确地预报天气、海洋和工业风向、太阳活动等。

理机系统)或两者都使用。这种计算机中的并行处理不仅出现于各种超级计算机中,而且日益遍及各种工作站中的并行处理(如 Sun 公司生产的 Sun 工作站)。

并行计算机的主要功能结构和特点可概括为:

- 1)流水线结构 其特点相当于工业生产中的装配线技术,改进运算和控制部件的功能。
- 2)功能结构 为执行不同的功能提供几个独立部件,例如逻辑部件,加法或乘法部件,并允许这些部件同时处理各种数据。
- 3)阵列结构 在统一控制下,提供一个很多相同的处理部件组成的阵列,这些部件同时执行相同的操作,但是与其相对应的专用存储器中的数据是不同的。
- 4)多道处理结构 提供  $n$  台处理机,但各台处理机只执行其各自的指令,通常经过一个共同的存储器相互通信。

在众多的并行计算机中,某些特定的计算机可能兼有上述并行特点的部分或全部。例如:一个处理机阵列可以把流水线运算部件作为它的处理部件,同时在多部件计算机中,某一功能部件可以是一个处理机阵列。

### § 1.1.2 计算机与算法的分类

计算机按其指令流与数据流的执行方式可分为四类:

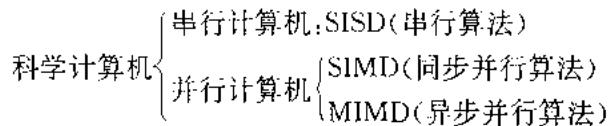
(1)SISD 计算机 即单指令流单数据流计算机。这类机器就是常用的(冯·诺依曼型)计算机。为这类机器编写的程序都是根据一种“串行算法”编制的。

(2)SIMD 计算机 即单指令流多数据流计算机。如阵列式计算机:ILLIAC IV,流水线计算机:国产 757 机与银河(YH-I,STAR-100,CYBER 205,Cray-1,ICL-DAP,IBM 360/91 等都属于 SIMD。它们都要求使用一类“同步并行算法”。这类算法基于进行向量与矩阵运算时各分量的高度同步并行性,故这类计算机也称为向量机。自从 1972 年并行计算机系统 ILLIAC IV 与 1974 年 STAR-100 投入运行以来,同步并行算法已有较多的研究。特别是 Cray-1 与 CYBER 205 等类型的计算机已批量生产,因此与之相适应的同步并行算法的发展也较快。我国在研制银河机的同时也开展了同步并行算法的研究,并取得了可喜的进展。

(3)MIMD 计算机 即多指令流多数据流计算机。多处理机与多计算机就属于这种类型。如我国的银河-II(YH-II)多处理机系统、武汉大学的 WuPP-80 并行处理系统、Carnegie-Mellon 大学的 Cmmp 与 C'm 系统、Loughborough 大学的 LuT<sub>2</sub> 与 LuT<sub>4</sub> 等多计算机系统、California 工学院的 Hypercube、Rochester 大学的 Butterfly™ 等多处理机系统,还有 IBM 公司的 ICAP 系列、Wisconsin-Madison 大学的 CRYSTAL,以及多向量机(M SIMD 或称 Pipelined MIMD)HEP、Cray X-MP 系列、Cray-2 等等。这类机器的并行运算过程通常都具有高度的“自治性”,即异步地并行运算。它要求一类异步并行算法的支持。MIMD 计算机在科学计算方面的运用尚处在试验阶段,没有公认的计算机模型。因此异步并行算法的研究也处在试验阶段,但这类计算机有较高的性能价格比、灵活的系统可扩充性、良好的实时性、可靠性与容错性等优点。因此,作为新一代计算机的雏型,各式各样的 MIMD 试验系统越来越多,与之相适应的异步并行算法的研究成果也越来越多,越来越引人注目。

(4)MISD 计算机 即多指令流单数据流计算机,这种机器很少用于科学计算。

综合上述,目前用于科学计算的计算机与算法可粗略地分类于下:



我们强调上面的分类是粗略的,这样对研究算法设计有好处,易于抓主要矛盾,尤其对初学者。但实用上,由于应用问题千差万别,计算机结构五花八门,并行算法尚未完善,故现在的并行计算机上实际运行的是“串、并”混合算法。正因为如此,并行计算机的性能并未最有效地发挥。

由于 SIMD 计算机只在数据流上有并行性,而指令流仍然是串行的,故有人认为这类计算机算法(只有数据流并行)仍属串行算法,或称为向量化算法,用以区别具有二维并行性(数据流并行,指令流并行)的 MIMD 算法。从算法发展历史来看,人们都称 SIMD 算法为并行算法,只是今天 MIMD 计算机大量涌现之后才出现了更细的划分。因此我们认为,作为粗略分类以区别一维并行性(数据流)与多维并行性,还是分别称为同步并行算法与异步并行算法为好。至于 MIMD 计算机和 SIMD 算法也有同步与异步之分。从其所含处理机之台数来说又有浅度并行(10~100 台)、深度并行(100~10000 台)与极度并行(10000 台以上)之分;从进程的粒度来分有高级并行(数学模型级)、中级并行与低级并行(算术表达式级)之分;从存贮的方式上又有共享存贮与分布式存贮之分;从机器的连接与通信方式等方面又可分成许多种。所以,机器分类越细,算法分类也越细,分类过细反而不易抓住主要矛盾。

### § 1.1.3 并行处理机的几种形式

尽管并行处理的概念可以追溯到现代计算机的发明之前,但并行处理作为一门技术被深入地研究与普及应用却是近十多年的事情,其蓬勃发展更是近几年才出现的,而并行处理机要像传统冯·诺依曼结构那样被广泛而有效地应用则需要进一步研究和推广。

并行处理机有三个基本结构类型:流水线结构 SIMD 计算机、阵列结构 SIMD 计算机和多道处理结构 MIMD 计算机。

#### 1. 流水线结构 SIMD 计算机

流水线结构是基于时间重叠原理,把每条指令分成若干个操作(如取指令、译码、取操作数、执行和写结果),每个操作分别由不同的部件执行,以此达到同时处理多条指令。对每个部件而言,每条指令中的相同操作像流水线一样被连续处理,从而实现时间重叠的并行计算。当流水线不断流时,实际的执行速度相当于流水执行部件一级通过的速度,即机器周期。

许多早期开发的并行处理计算机,就是流水线向量处理器。在 50 年代末,由英国曼彻斯特大学设计和制造的 ATLAS 计算机就是一种早期的使用流水线计算机。一条指令的执行被分为四段(指令提取、操作数位置计算、操作数据提取和算术处理),该阶段可以制成流水线,于是在理想状况下,指令 1 的第 4 段,指令 2 的第 3 段,指令 3 的第 2 段,指令 4 的第 1 段能够被连续运行,要使用好这种设备,必须有一种“远见”,以便确定哪些指令可以联结而不致于影响程序的逻辑意图。

向量流水线是向量操作数和向量指令流水线的意图之简单延伸。向量运算之所以很适合流水线作业的原因是它们在其分量上形成一种可预见的正规的一列标量操作。作为一个简单的例子,设  $a + b \rightarrow c$ ,这个加法操作可以划分为三个阶段,每个阶段需要一个时钟循环

来执行。于是：

- 在第一个时钟循环时：  $a_1$  与  $b_1$  在加法第一阶段；
- 在第二个时钟循环时：  $a_1$  与  $b_1$  在加法第二阶段；  
 $a_2$  与  $b_2$  在加法第一阶段；
- 在第三个时钟循环时：  $a_1$  与  $b_1$  在加法第三阶段；  
 $a_2$  与  $b_2$  在加法第二阶段；  
 $a_3$  与  $b_3$  在加法第一阶段。

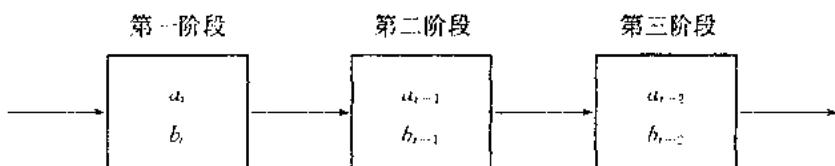


图 1-1 第  $i$  个时钟循环的流水线形态

这以后，所有的这三个阶段都连续地工作，直到进入  $a$  与  $b$  的最后一个分量。第  $i$  个时钟循环的示意图如图 1-1 所示，结果是： $a_{i-2} + b_{i-2} \Rightarrow c_{i-2}$  加法完成。

## 2. 阵列结构处理机

一个阵列结构处理机(或数据并行处理机)是属于 SIMD 计算机。阵列结构是基于空间里的重复原理，将许多具有相同功能和处理能力的处理器按一定的拓扑结构互相连接而构成的，在这种结构中，各个处理部件在单指令流的控制下并行处理各自的数据流，因此特别适用于向量和矩阵处理。在阵列机中，每个处理器都要同等地担负多种运算功能，其复杂程度视需要而定，可以是简单的一位微处理器，也可以是功能强大的 32 位微处理器。阵列机的专用性很强，其性能与算法结构有着密切的联系，对于某些特定用途(如信号处理、图像处理和电路模拟等)而言，可以达到超高速运算。例如，由 16384 个一位处理单元按  $128 \times 128$  阵列结构组成的大规模并行处理机(MPP)，对 8 位整数加法速度可达每秒 65 亿次；由 65536 个一位处理单元组成的 CM-2 的运算速度可达每秒 25 亿次，对特殊问题可达每秒 70 亿次。1995 年我国通过鉴定的曙光 1000 大规模并行计算机系统(MPP)，突破了大规模并行处理的关键技术，采用了多项 90 年代的国际最新技术。特别是它的蛀洞路由器芯片和并行优化编译器，在世界上堪称是最先进的技术。曙光 1000 由 36 个结点处理机组成，其中 32 个计算结点机，2 个服务结点机，2 个输入输出结点机，采用  $6 \times 6$  的网格结构。曙光 1000 的峰值速度可达每秒 25 亿单精度浮点运算，求解线性方程组的速度可达每秒 11.2 亿次浮点运算。还有 ILLIAC IV, DAP 及 IBM GF-11 和 TF-11 等都属此类计算机。

思维公司推出的换代产品 CM-5，集 SIMD 和 MIMD 优点于一身，是目前数据并行计算机最成功的产品。

CM-5 体系中最重要的创新就是它的体系结构超越了 SIMD 和 MIMD 两者。既有 SIMD 编程的简易性，又有 MIMD 执行的灵活性。它提供了使机器的所有部分都恰到好处地保持协调同步的硬件和软件的结合。CM-5 的构造块是标准的 RISC 处理器另加上了一些并行处理部件，它们是一个特殊的存储控制器、一个通信接口，并且为浮点运算增加了 4 个 64 位的向量单元(总的峰值能大于 100Mflops)，两个称为控制网络和数据网络的通信网络

把这些处理器连结起来。这两个网络起着系统总线的作用，你可以把处理器、存储器和 I/O 接到这个“总线”上面。但是，和通常的系统不同，它的能力随着连到它上面的设备的增加而增加。譬如，连 1000 个处理器，这个网络就具有每秒钟传输 50Gbits 的通讯能力。

数据网络是用来传输数据的电话交换系统，它使每个处理器都能存取其它处理器的存储器。因此，所有的数据都能被所有的处理器存取。控制网络是解决 SIMD/MIMD 问题的关键。这个网络能够把指令同时扩散到所有的处理器，并且建立同步。所有的处理器都在做同一件事的时候，这个网络的作用是从一个中心位置向其它处理器扩散程序快。这时它的操作很像一台 SIMD 机器，处理器的时钟的锁相环保持严格的同步。但是，当这些处理器要执行不同的作业时，它们就开始从各自的高速缓冲存储器中取指令了，这些高速缓存器是由操作系统自动加载指令的。一旦这些处理器需要一起做同一件事的时候，它们就又被控制网络同步起来。编译程序会识别什么时候要这样做，并且自动生成相应的程序。

就用户来说，处理器处理执行程序、处理器的同步以及程序的扩散、传播等所有这些事都是透明的。用户只要写一个程序，余下的所有细微事宜都会由编译操作系统和控制网络硬件来处理，从而形成既有 SIMD 机的编程简易性又有 MIMD 机的灵活性的操作系统。

### 3. 多道处理结构 MIMD 计算机

MIMD 并行处理计算机由多台具有独立处理能力的处理机按照一定的互连模式相连，其中每台处理机都有自己的指令流和数据流，可以各自执行独立的程序，也可以通过互连网络进行数据和信息交换，协同完成一个大问题的处理任务。根据其存储结构和通信机制，MIMD 计算机又可分为四类：即集中存储/共享变量、分散存储/信息传递、集中存储/信息传递和分散存储/共享变量。其中集中存储/共享变量方式的 MIMD 计算机通用性强，编程环境和软件继承性较好，适合于通信量较大的问题，它不仅是目前并行机和并行工作站的主流技术，而且也是各国研制超级计算机的主攻方向。分散存储/信息传递方式 MIMD 计算机是近几年进入市场的超立方体结构多处理系统，其主要特点是性能价格比高，不需要非常高速的互连网络，网络连接也较低，很适合于事件驱动的模拟应用及科研部门和大学作为研究并行算法的工具，也是构成由成千上万台处理机组成的“高度并行”计算机的较理想的结构。但是，缺乏软件和需要重新编程是这类计算机目前所面临的最大难题。关于集中存储/信息传递和分散存储/共享变量也取得了一定的成效。

#### § 1.1.4 程序语言

在后续章节里我们将讨论并行计算机的并行程序设计语言，同时讨论适合多道处理机系统的扩充并行语言。

#### § 1.1.5 性能测量

比较各种计算机系统的性能，就必须测量其速度。我们已经提到 Mflops 作为量化一个浮点单元速度的一个测量，性能测量一般用 mips (millions of instructions per second 每秒运行数百万次)。值得一提的是从 flops 到 mips 的转换是依赖于机器本身的，在一个标量处理机上，一个浮点运算将包含两到五个指令。

制造商引用的性能图表一般提到：

- 峰值速度：计算确定在一个循环时间里处理机可能运行的指令数量。

- 持续速度: 像典型应用一样的一种速度测量。

峰值图与持续图可以看作一个系统的一个分量的整体或部分, 前者表示这个系统性能的一个上限, 而后者则像通常的一种性能测量。它往往仅是前者的一个分数。

除主存储器外, 现代的一些处理机相对要小些, 速度要高些。

性能测量只是在总体上给出一个尺度, 检测一个计算机是好是坏, 要看它的应用场合。

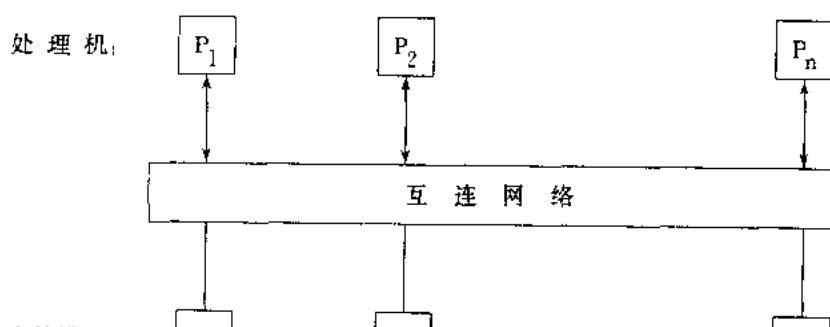
## § 1.2 多道处理机系统

多道处理机系统包括一些相互联结的处理机, 它们每个都能单独地完成复杂的工作任务。一个单独的处理机或节点, 可以是标量或向量处理机, 甚至是一个处理系统。

若从存储和软件的角度考虑, 多道处理机又分为共享存储系统和局部存储系统:

- 所谓共享存储系统是指这种系统里的主存储器是多个处理机都可使用的全局资源;
- 所谓局部存储系统是指这种系统里的主存储器仅供单个处理机使用。

例如图 1-2 及图 1-3。Encore Multimax 和 Intel Hyper Cube 分别是共享存储系统和局部存储系统。Multimax 系统的处理机仅含有标量处理特点。Hyper Cube 系统里的处理机有无向量处理能力依赖于特定的模型, 本节将详细讨论多道处理机 Multimax Hyper Cube 系统。



### § 1.2.1 互连

处理器与存储器之间硬件上的互连有点像总线驱动器,或者处理机本身就有交流的能力,能使它们直接地互连起来。一般使用下面两种部件开关。

一种是纵横开关,根据组合结构的需要,可以互连多道进口与多道出口。如一个 $2 \times 2$ 纵横开关就有2道出口和2道进口(如图1-4)。开关允许进口*i*与出口*i*相连,同时允许进口*i*与出口(*i*+1)相连,而广播方式是把每个进口传输到各个出口。

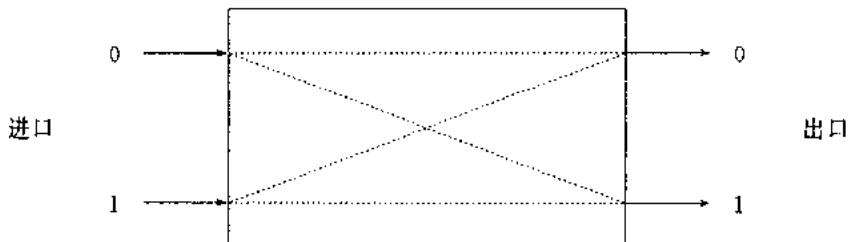


图1-4 A $2 \times 2$ 纵横开关

另一种补充互连的方式是建立一个多级开关,其中每一个分支是一个纵横开关。例如:四级进口与四级出口可以通过一个二级开关系统来实现,它里面包含两个 $2 \times 2$ 纵横开关。参见图1-5。

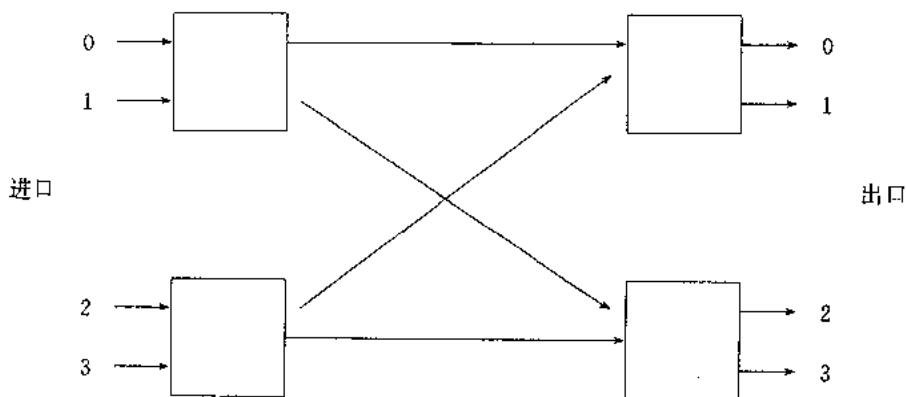


图1-5 A $4 \times 4$ 多级开关组成的 $2 \times 2$ 纵横

### § 1.2.2 共享存储器系统

存储系统的组成块是处理元素,存储模型及一些连接到存储器指令系统,这些综合的模型也有许多不同,如每个处理机可能有标量处理与向量处理性能的不同。指令网中也有纵横开关与多级开关的不同,这里所考虑是通常系统中的一些例子。我们注意到 Encore multimes 的构造,因为我们将介绍这个系统的某些性能,除此以外,Multimes 还提供一些在并行机上的更广泛的合适模型。

#### 1. 向量巨型计算机

向量巨型计算机最初的发展是Cray计算机的研制和生产。Cray-2(1984/5)有四个向量处理机,每个处理机有256M个字节的存储量。而 LINPACK 有 1.4Gflops,Cray Y-MP

(1987/8)也是一个多道向量处理机,它有 8 个处理机,而一个 LINPACK 有 2.1Gflops。

从价廉物美的角度看,要首推日本生产的向量巨型计算机 NEC SX-3/14 与 FUJITSU VP 2600/10,它们各自带有一个处理周期分别为 2.9 毫微秒和 3.2 毫微秒的处理机,并有 LINPACK 标准超过 4Gflops。

在 1991 年,Convex 推出了 C3800 巨型计算机,它有 8 个处理器,4Gbytes 存储器,一个 0.1Gflops 的 LINPACK 标准处理器,8 个 LINPACK 标准的计算机达到 1Gflops。

Alliant FX/80(1987)是一个小型的超级计算机,带有8个处理机,每个都具有向量处理能力,每个处理元素通过一个纵横开关连接到一个512Kbytes的超高速缓冲存储器(cache),它的带宽为376Mbytes/s,超高速缓冲存储器(cache)与主存储器通过两个188Mbytes/s的总线通信,峰值性能为188Mflops,LINPACK标准为69Mflops。

## 2. 对称共享存储器的多道处理机

速度为 8.5mips,且具有 256K 字节大小的写入延迟时间,仅在需要时主存上相应的值才会被修改。这种写入延迟的超高速缓冲存储器可以帮助减少总线的流通量。

总的来说,320 系统可以进行 16mips,520 系统则能进行 119mips。

320 系统主要是用于教学体系,基本上所有的执行程序并不十分要求并行化处理,通常把它们作为 Pascal 程序队列处理。

520 系统主要用作研究型的机器,也可以是一个多用户源。支持语言有 C, Pascal 和 Fortran 77, 用这些编写的程序能动态地生成并行的任务,除了 Fortran 以外,这些任务的数量不受有效处理器的实际数量的限制。一个多任务库把任务分组输入进程,然后进程由系统软件分配完成在用户控制之外,它能显示需要多少处理机。

### § 1.2.3 局部存储器系统

局部存储器系统的组成部分是多道处理元素,每个都带有它自己的主存储器替代系统,它们的不同之处是指令系统的不同,下面是几个例子。

#### 1. 超立方体

我们定义一个 0 维的超立方体为一个单个处理机,建造更高维的超立方体可以通过连接两个低维的超立方体来实现。在图 1-6 中我们将分别看到 0,1,2,3 维的各种超立方体图示。可以看出维数为  $p=2^i$  的超立方体,可以通过连接每个节点而连接到  $i$  个邻结点,两个结点的最大距离为  $i$ (即  $\log_2 p$ )。

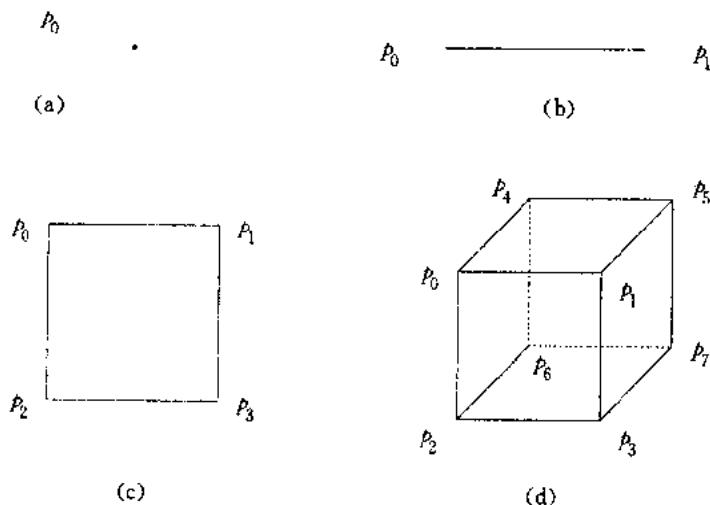


图1-6 维数分别为0, 1, 2, 3的超立方体

现有的超立方体样机有 Caltech Cosmic Cube (Seitz 1985) Caltech (JPZ), Mark II (Tuazon Peterson et. al 1985) 推出了 Intel iPSC/1 超立方体及 AMETEK S14 超立方体。

#### 2. Intel 超立方体

iPSC/1 机有 128 个处理节点,每个节点包含一个 Intel 80286 处理器,有一个 80287 协处理器,信息通过一个个节点最后到达终点,所以超立方体节点的拓扑性质很重要。

iPSC/2 也有 128 个节点,每个节点包含一个 Intel 80386 处理器。有一个 80387 协处理器,有 16 Mbytes 的存储器,这种协处理器可以被一个 Sx 处理机替代,它也是一个高速的标

量处理协同处理器。

用户最关心的 iPSC/2 最有意义的改进,是在硬件上提供的直接连接模块。这些通常的功能,使得一个节点能将信息传递给任何其它的节点,这些信息经过内部节点的通信模块,但不干扰那些节点处理器。一般的使用者将计算机看作完全连接节点的结构,而不必顾及超立方体节点的拓扑结构。另外一个有意义的改变是超立方体的分离成为可能,从而使若干个用户能使用这个超立方体的一部分。

iPSC/860 与 iPSC/2 相类似,节点处理器是 Intel i860 处理机,这种处理器可以通过使用先进的通道技巧达到高浮点标量和向量的操作。每个节点的存储量是 8 Mbytes。

iPSC/2 和 iPSC/860 都装备一台 SRM (System Resource Manager) 前端处理器,SRM 被用来编辑节点程序,并控制这个超立方体的运行。

### 3. Daresbury 设备

以安装在英国科学工程研究院的 Intel 超立方体机器 Daresbury 为例,该机器可以使用一个 32 个节点的 iPSC/860 和 32 个节点的 iPSC/2,并带有标量和向量双重节点。每个机器有自己的 SRM。在 Daresbury 中这些 SRM 有一个局部区域网络和其他的机器相连接,其中的一个 Convex C<sub>220</sub> 功能是作为一个超立方体的远程主机,以减轻 SRM 的工作负担。

超立方体被用来专门作为研究的机器,开发 Intel 立方体共享功能,提供给多用户服务。使用的是 Fortran 77 和 C 语言。

### 4. 晶片计算机系统

迄今为止,我们所考虑的所有多道处理机系统(共享的和局部存储),是由标准晶片构造的。它包括一个处理器,一个存储模块,一个外部存储接口和四个连接控制器。

所有这些公用总线(图 1-7)每一个线路控制器支持一条外线。它通过硬件将晶片与计算机连接起来,或者是间接连接。

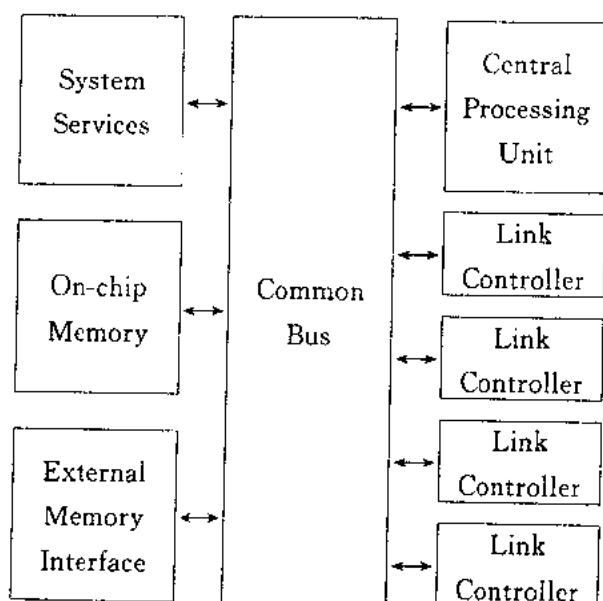


图 1-7 晶片计算机逻辑图

第一代晶片计算机(如 T414, T212, T800)连接速度为 10Mbits/s。而现在通用的连接翻

译程序每秒 20 Mbits/s。Inmos 引用了一个双方向的数据库, 对每一个连接可以多达 2.35Mbytes/s。

目前的晶片计算机包括以下几种:

- T414

1985 年引入 T414, 宣告了晶片计算机的诞生。作为一个 32 位的机器, T414 可以带有 2K 字节的存储。T414-20 能够在 20MHz 运行, 还有一个外部带有 50ns 的时钟速度。

- T212

T212 与 T414 类似, 其不同是客观存在的是 16 位机器。

- T800

T800 包含了晶片处理系统, 一个附加的晶片为 64 位的浮点处理器, 这个浮点处理器与共用总线相连接, 而且可以独立运行, 其峰值速度可以分别达到 1.5~2.25Mflops。

因为晶片计算机有四个链式, 所以可能有别的布局, 例如带环形的矩形网络, 如图 1-8。注意, 由于这些环形用到了所有的有效链式, 因此有必要增加一个晶片计算机去连接环形和根。

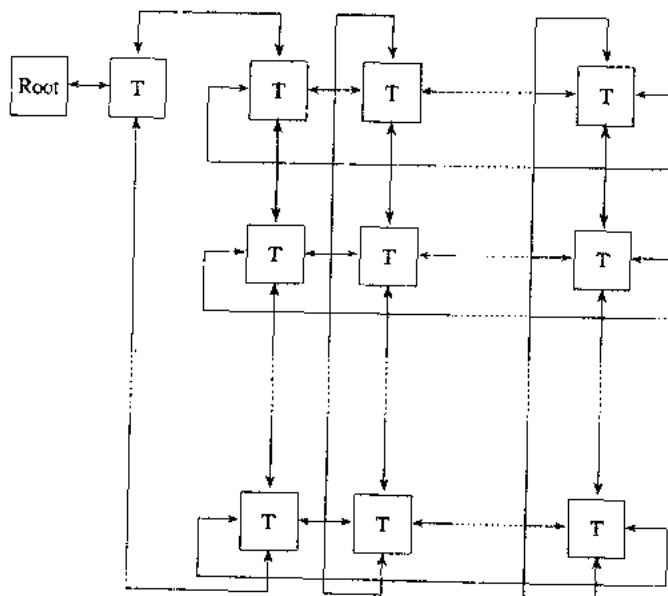


图 1-8 矩形网络晶片机

#### § 1.2.4 对局部存储系统的共享

考虑到后面章节的需要, 在此先对两类系统作一基本比较。

- 汇编

对许多程序语言, 如 Fortran 77, C 和 Pascal(带有并行支持), 汇编对两类系统总是有用的。支持真并行的 Ada 汇编也有利于某类多进程。更进一步地, 自动并行序列码的汇编非常有利于共享存储系统, 对局部存储系统编码和数据都有必要进行划分。

- 启动系统

单局部存储系统的初始费用可能是完全合理的,因此包含大量额外进程的升级费用也相对便宜。比较起来,共享存储系统的项目开销很高,因此升级相应地开销大。

- 扩展

已考虑的共享存储系统中处理部件数目相当少。作为补偿,每一处理部件可望相当有效。在任何情况下,能用的处理部件都有物理上界,这个上界产生于交互设计的限制。例如,在毫微总线 Encore Multimax 上刚好有够用的时间片供给 20 个处理机。从理论上讲,局部存储系统对能用的处理机数无理论极限,虽然在系统的极点端处理机间距离可能变得高而不可接受。我们称由非常多处理机构成的系统为大容量并行。局部存储系统消耗单元少而能量无限扩张,其软件支持被迅速改善,进展正有利于局部存储系统。它们带有处理机间通信,这些处理机对共享存储系统来说足够快。

### § 1.2.5 处理机与进程

重要的是需要区别“处理机”与“进程”。前者是一功能单元(硬件),后者是关于数据操作的组合(软件)。在单处理机上,只有一个进程在任一时刻要处理,但可能有多进程时间共享。在多处理机系统,这可能造成结果的混乱。在多数情况下,将情况假定为处理机与进程间一一对应。即:在一个带  $p$  个处理机的多处理机上有  $p$  个进程,每个处理机有一个进程且时间共享。因此,在局部存储系统,交互进程通信将包含沿交互进程链送往的信息。

## § 1.3 度量程序操作

在 § 1.2 讨论了多处理机系统的例子并且对操作作了分析,下面进一步作一些说明。

### § 1.3.1 粒度

对算法内在的并行性进行分类,常用的度量是粒度,它用于进程而非处理机。它是一个指令的数字,这些指令在某种同步指令需要发生前可以并行地操作。粒度是一量词术语,特别地,我们可视并行为以下三种情况:

- 细粒度
- 中粒度
- 粗粒度

分别根据同步点间指令数是单位阶、十阶或百阶而划分类型。SIMD 机最适于细到中等粒度并行,而 MIMD 机更适合中至粗粒度并行。

与粒度大小有关的是计算/通信率。比率大意味着大粒度。

### § 1.3.2 加速与效率

设  $T_p$  是关于  $p$  个处理程序的并行程序的执行时间,有下列定义:

- 关于  $p$  个处理程序的加速率  $S_p = T_0/T_p$

这里  $T_0$  是关于单处理程序的最快串行算法所需时间。应该注意,加速率是与对给定问题的最快串行算法的并行算法对应的。它反映了从一带有单处理机的串行机到带有  $p$  个理想处

理机的并行机的应用中所获得的利益。

- 关于  $p$  个处理程序的算法加速率  $\bar{S}_p = T_1/T_p$

该量用以度量加速，此加速由一给定算法之并行获得。它直接度量关于并行算法操作的同步和通信延迟的影响，本书用这个加速率的定义而不作进一步限制。理想地，我们希望  $\bar{S}_p$  线性增长，其斜率为 1。不幸的是，甚至对一个“好的”并行算法，我们至多能期望初始加速接近于线性增长，而后逐渐地随处理程序饱和而减弱。

- 关于  $p$  个处理程序的效率  $E_p = 100 \times \bar{S}_p/p$

注意效率按百分比度量且在理想状态下我们希望对所有  $p$  有 100% 的效率。实际中我们希望效率随  $p$  的增加而减少。

不难看出，正如所定义的那样，加速与效率均按处理程序数目度量。

加速与效率显然是并行算法发展的驱动力。但是，一定不要忘了其他特征，比如精度与依赖性。

### § 1.3.3 Amdahl 法则

如上所述，理想状态是加速率随  $p$  成线性增加，斜率为 1，给一个 100% 的斜率。

Amdahl 法则 设  $\gamma$  是一个并行程序的分数，而  $S = 1 - \gamma$  是剩余的串行分数，那么对  $p$  个处理程序算法加速率  $\bar{S}_p$  满足

$$\bar{S}_p \leq \frac{1}{S + \gamma/p}$$

这个结果的证明是直接的。用时间来表示串行部分残留常数( $ST_1$ )，对时间而言，倒过来随  $p$  降低，算式为  $\gamma T_1/p$ 。

所以

$$T_p \geq ST_1 + \gamma T_1/p$$

且

$$\bar{S}_p \leq \frac{T_1}{ST_1 + \gamma T_1/p} = \frac{1}{S + \gamma/p}$$

当考虑算法的并行时，Amdahl 法则似乎有深刻的结果，因为它对加速率强加了一个上界。

例如，如果程序仅 50% 是并行的，那么  $S = \gamma = \frac{1}{2}$ ，

所以

$$\bar{S}_p \leq \frac{2}{1 + 1/p}$$

所以当  $p \rightarrow \infty$  时  $\bar{S}_p \leq 2$ 。因此，对这个特殊程序，加速率被限制为 2，不计用到的处理程序数。

### § 1.3.4 负载平衡与吞吐量

当希望预测应用程序有效率时，关键是证实所有处理程序正在尽可能多的时间内做有用的事。也就是说，我们要尽可能地避免并发时刻的处理程序在其他处理程序处理前等待信息。虽然在这种意义下，运行在单处理机下的序列程序 100% 的有效。目标是以这种方式改善算法，以尽可能多地用到处理程序，而在同时使所有处理程序充分发挥作用。我们称此为负载平衡。

## 第二章 并行算法概述

并行算法在体系结构方面的一个重要特点就是“并行化”。计算机发展的这种“并行化”趋势，必然会引起数值计算方法的“并行化”。在并行计算机上进行科学计算必须有并行算法支持，否则，机器纵然身怀绝技，也难以大显神通，必然造成计算机资源的莫大浪费。而并行算法的发展，反过来又会促进计算机结构的各种新的并行形式出现。事实上，现在正在研制中的许多并行处理机系统就是算法、软件、硬件三结合的产物。

### § 2.1 并行算法发展的几个阶段

并行算法是随着并行计算机的发展而发展起来的一门学科，它的发展大致经历如下三个阶段：

#### 1. 并行计算机前期(1972年以前)

1972年以前，是并行算法的预研期。这期间由于大型科学计算的需要，特别是解偏微分方程的需要，人们研制了新型结构计算机，一种是陈列式结构，一种是流水线结构。它们都是为适应求解差分方程组或线性代数方程组迭代算法提出的。在这一时期，并行算法的研究建立在理想化的并行计算机模型上，其假设是：

- (1)所有处理机是相同的；
- (2)处理机台数是无限的；
- (3)有任何大的存储器可供使用，并且同时为所有的处理机同时存取；
- (4)四则运算均花同一时间单位；
- (5)存、取、同步和通信等不计时间。

建立在这种理想化基础上的算法是一种理想化的同步算法，而且绝大部分并行算法研究都以 SIMD 计算机为背景。这时期，都用下面的公式度量算法的“加速”(Speed up)，与效率(Efficiency)：

$$\text{加速} \quad S_p = T_1 / T_p \quad (\geq 1) \quad (2.1)$$

$$\text{效率} \quad E_p = S_p / p \quad (\leq 1) \quad (2.2)$$

其中  $p$  为处理机台数， $T_1$  为串行算法计算某个问题所需要的时间单位(或时间步)， $T_p$  为并行算法计算同一问题时所需的时间单位。

这里要注意的是：执行串行算法的单处理机与执行并行算法的  $p$  台并行处理机都是同样的处理机(假设 1)，而其它操作又不计时间(假设 4)。上述公式当时能为人们所接受是因为在这种假设下， $T_1$  就是一台处理机执行的时间， $T_p$  就是用  $p$  台处理机执行的时间(理想的)。

如计算两个矩阵相乘

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & & & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} \quad (2.3)$$

其中

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj} \quad (2.4)$$

如果有  $n^3$  台处理机, 其步骤如下:

第一步: 每台处理机从内存中各取出相应的元素  $a_{ik}, b_{kj}$ ;

第二步: 每台处理机作乘法  $a_{ik}b_{kj}$ ;

第三步: 用一部分处理机对它们求和;

第四步: 将  $c_{ij}$  的结果写入内存。

由假设, 第一、四步不花时间; 第二步花一个时间单位; 第三步花  $\log_2 n$  个时间单位, 故

$$T_n^3 = 1 + \log_2 n$$

若用串行算法, 则第二步要作  $n^3$  次乘法, 第三步要作  $(n-1)n^2$  个加法, 故

$$T_1 = 2n^3 - n^2 \quad (2.5)$$

$$S_n^3 = (2n-1)n^2 / (1 + \log_2 n) \quad (2.6)$$

$$E_n^3 = (2n-1)/n(1 + \log_2 n) \quad (2.7)$$

如果限制处理机台数, 比如  $p=n$ , 则  $S_p < S_n^3$ , 而  $E_p > E_n^3$ 。由此看来,  $S_p$  和  $E_p$  有时是两个互相矛盾的目标: 若要得到最大加速的算法, 可能就要以降低算法的效率为代价。

在这一时期, 递归计算问题是 SIMD 并行算法设计的一个难点, 许多文献集中在这类问题上。值得提出的是, 1969 年 D. Chazan 与 W. Miranker 的关于“混乱松弛法”(Chaotic relaxation)的文章为 MIMD 计算机算法做了开创性工作。

## 2. 并行计算机的初期(1972~1981 年)

并行算法的研究进入了实践阶段。首先是并行机前期的并行算法得到了检验而被重新估价, 新的算法大都带有具体计算机的烙印。尽管都是向量机, 但它们要求被计算的向量长度, 向量数据的存储方式等都不相同。同是 SIMD 计算机, 它们对并行算法也有不同的选择与评估标准。

经过实践的检验, 人们发现: 理想并行算法的公理系统与实际大为脱节, 关于计算复杂性研究也有问题。例如前面谈到的矩阵乘法, 其中每一步的取数和第四步的存数是不可忽略的。在第一步, 数组 A, B 的每个元素必须同时为  $n$  台不同的处理机所索取。对于单进出口存储器(a single-port memory)并行机, 其处理机的每个地址同一时间仅允许一台处理机取数。这样一来, 第一步“取数”, 就需要  $n$  个单位时间, 故取数据共需  $n^3$  个时间单位。

如果将  $T_1$  改为单台处理机上执行最佳算法解题的时间,  $T_p$  表示在  $p$  台处理机的并行计算机上执行并行算法解同一问题的时间, 则并行处理的“加速”为:

$$S_p = \frac{\text{在单机上用最快的串行算法执行时间}}{\text{在 } p \text{ 台处理机上用并行算法执行时间}}$$

这一定义当时为大多数人所接受。

由于“最快的串行算法”也没有定义, 例如解  $n$  阶线性代数方程组的“最快串行算法”是什么, 至少目前尚无定论, 只有解一些简单的问题才有公认的答案。如解一般三对角方程的

追赶法等。

该阶段许多用户投入到并行算法的研究中去,如 NASA,LLNL 与 LANL 中的大型科学计算的物理学家、空气动力学家、结构分析学家、计算数学家都转入并行算法的研究。在这阶段里许多同步算法形成了相应的软件。NASA,LLNL 和 LANL 在 Cray-1 上的许多并行应用软件与标准程序正在 SIMD 机上为广大用户所用。

综合上述,这时期并行算法的研究特点是:

- (1) 主要研究同步并行算法,特别是向量式 SIMD 计算机算法;
- (2) 并行算法的计算复杂性分析与性能评估有了新的内容;
- (3) 并行算法与具体机器的相关性增加了;
- (4) 同步并行算法正逐步软件化;
- (5) 从事并行算法研究人员增多,成分复杂了(不单是计算机科学家)。

这时期由于美国卡内基-梅隆大学的 MIMD 计算机系统 Cmmp 问世,异步并行算法第一次得到了实践的机会,并行算法出现了新方向。

### 3. 并行计算机全盛期(1982 年以后)

这一时期,并行机结构百花齐放。1982 年以 Cray-1S 为处理机的多机系统 Cray X-MP 出现,并于 1984 年在 LLNL 等实验室投入使用。1983 年第一台商品化的 MIMD 计算机出现,Denelcor 公司的 HEP 批量生产。1985 年 Cray 研究公司又推出 Cray-2,它是由 4 台改进型的 Cray-1S 紧密耦合而成的 MIMD 计算机。它有较好的算法基础与较多的应用软件,但完全成功还有待于研制新的异步并行算法、并行语言以及完善它们的系统软件。

在这一时期,超立方体(Hypercube)计算机,脉动(Systolic)计算机以及各种未定型的试验系统如雨后春笋般兴起。

在这一时期还发展了小型超级计算机,如 Gould 公司研制的具有 8 个处理器的小型超级计算机 NP1/180,运行速度可达 300Mflops;Alliant 公司研制的由 8 个单元组成的小型超级计算机 FX/8,其运行速度可达 94Mflops。我国武汉大学的并行处理机系统 WuPP-80 由 4 台微机紧密耦合而成,1982 年投入运转,在上面试验了许多 MIMD 算法。

这期间,并行算法的研究异常活跃,有围绕 SIMD 机的同步并行算法研究;有围绕着 MIMD 机的异步并行算法研究;有针对脉动计算机的脉动算法的研究;还有为各种专用机而展开的并行算法研究等等。总之,这期间并行算法的研究是百花齐放,但研究的特点可归结为以下几点:

- (1) 同步并行算法更加成熟与软件化;
- (2) 异步并行算法的研究非常活跃;
- (3) 算法分类与复杂性分析被重新考虑;
- (4) 算子分裂法、系统分裂法和区域分裂法的研究成为热闹课题;
- (5) 卷入并行计算研究的人越来越多,可以说它已成了高科技中竞争最为激烈的领域之一。

怎样评估并行计算机的速度?又如何评价它的性能呢?1983 年 LANL 的 I. Bucher 推广了 Amdahl 法用来评估一个算法在并行处理机系统上的性能。Bucher 公式说明了一个算法执行的时间是与下述量成正比的:

$$(F_1/S_1) + (F_2/S_2) + (F_3/S_3)$$

其中  $S_1, S_2$  与  $S_3$  分别为该并行机处理串行程序, 向量程序(同步并行算法)和异步并行程序的速度, 而  $F_1, F_2$  和  $F_3$  分别表示一个算法在计算一个问题时只能在一个处理单元上串行处理, 同步并行处理与异步并行处理所占荷载的百分比。例如在 FX/8 系统上, 同步速度(并行向量计算速度)是单机串行速度的 32 倍, 异步并行标量速度是单机串行速度的 8 倍。若一个算法使 30% 的计算量能串行, 10% 可向量化执行, 60% 可异步并行执行, 则该算法在 FX/8 上计算该问题时执行时间为串行算法解该问题所需时间的 37.8%, 即

$$\frac{0.3}{1} + \frac{0.1}{32} + \frac{0.6}{8} = 0.3 + 0.0031 + 0.075 = 37.8\%$$

如果一个算法能使该问题的一半荷载可同步处理, 另一半可异步并行处理, 则执行该算法的时间仅为串行算法的 7.81%, 即

$$\frac{0.5}{32} + \frac{0.5}{8} = 0.0156 + 0.0625 = 7.81\%$$

由此可见, 并行计算机性能的充分发挥, 有赖于并行算法的研究, 而并行算法的研究又与并行计算机结构密切相关。

## § 2.2 同步并行算法

构造并行算法有两个原则: 一个叫做“分而治之”, 另一个叫做“重新编序”。分而治之是把一个问题分裂为一些可以独立地或相对独立地进行处理的小问题。这些小问题之间的“独立程度”越大越好, 最好是它们都彼此无关, 即完全独立。这样就可以减少甚至无需处理机与处理机之间的通信, 减少“通信”的开销; 也减少“同步”的开销。因为在多处理机上计算一个问题时, 若每台处理机负责计算一个小的“子问题”, 子问题独立, 则处理机之间无需联系而省去交换信息的开销。彼此不要等待, 省去了“同步”的开销。这样就能大大提高多处理机并行处理的能力。在并行计算中, 求解子问题的先后次序是十分重要的。为了减少通信, 加速收敛, 适应某种计算机结构的需要, 改变子问题处理的次序(即“重新编序”)有很重要的作用。

为了说明这些原则的应用, 我们用求解偏微分方程的“模型问题”来研究其并行算法。

考虑 Poisson 方程的 Dirichlet 问题:

$$(*) = \begin{cases} -(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) = f(x, y), & \text{于 } \Omega = \{(x, y) | 0 < x < 1, 0 < y < 1\} \text{ 内} \\ u = \varphi(x, y), & \text{于 } \Gamma \text{ 上} \end{cases}$$

其中  $f$  与  $\varphi$  为已知函数,  $\Gamma$  为单位正方形  $\Omega$  的边界。

我们取  $h=I/N$  为步长, 即  $\Delta x=\Delta y=h$ , 用五点差分格式将问题(\*)离散化为线性代数方程组

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = h^2 f_{i,j}, \\ i, j = 1, 2, \dots, N-1$$

其中  $u_{i,j}=u(x_i, y_j)=u(ih, jh)$ , 称为网格函数(见图 2-1)。取  $N=10$ , 则  $h=0.1$ , 若将方程组写成简单迭代形式(Jacobi 方法)

$$u_{i,j}^{(k+1)} = (u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}) / 4 + h^2 f_{i,j} / 4, \\ k = 0, 1, 2, \dots$$

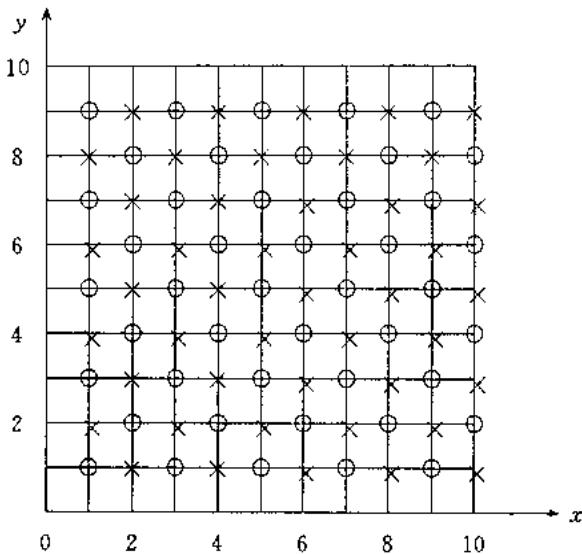


图2-1 网格图

其中  $u_{ij}^{(0)}$  为初始猜测, 写成矩阵形式有:

$$u^{(k+1)} = Au^{(k)} + b, \quad k=0,1,2,\dots$$

这样就将求解过程化成了矩阵乘向量  $Au^{(k)}$  和向量加法, 故 Jacobi 迭代法就是一种“向量算法”, 即同步并行算法。

若用具有  $(N-1)^2$  台处理机的 SIMD 计算机来解这个问题, 指定每台处理机计算一个点, 即一个方程, 则每迭代一步 ( $u^{(k)} \rightarrow u^{(k+1)}$ ) 只需 5 个时间单位 (4 次加法运算, 一次除法运算)。若迭代  $M$  步达到所要求精度的结果, 则并行计算时间为:

$$T(N-1)^2 = 5M$$

用串行计算机 (即一台处理机) 需要计算时间为:

$$T_1 = 6M(N-1)^2$$

$$\text{故加速 } (S_p) = \frac{5M(N-1)^2}{5M} = (N-1)^2$$

$$\text{效率 } (E) = S_p / (N-1)^2 = 1.$$

所以 Jacobi 迭代是一个效率高的同步并行算法, 第一台 SIMD 计算机 LLLAD N 就是基于这种理论的估计而设计的, 它的连接拓扑形式也是基于平面网络的这种布局。但是人所共知, 在串行机上解这种问题时, 大家不会用 Jacobi 方法而会用 Seidel 迭代与 SOR 方法。若单机用 Seidel 迭代, 则它的收敛速度要比 Jacobi 方法快一倍, 即迭代  $M/2$  次可达到所求的精确度。这时有:

$$T_r = \frac{5M}{2}(N-1)^2$$

$$S_p = (N-1)^2/2; \quad E = 1/2$$

如果考虑到机器间信息交换的频繁, 算法的实际效率要低得多。如果与取最佳松弛因子的 SOR 方法相比, 则 Jacobi 方法更理想。

设法调整计算的次序, 即先计算  $i+j$  为偶数的那些  $u_{i,j}$ , 后计算  $i+j$  为奇数的  $u_{i,j}$ 。先计算网格上打“○”的那些网点上的函数值, 再计算打“×”的网点上的函数值, 这种算法的收敛

速度与 Seidel 方法相同。人们称之为“奇-偶”松弛法，或“红-黑”(R-B)松弛法。如果引进最佳松弛因子，收敛速度将更快，如用  $(N-1)^2/2$  台处理机的 SIMD 计算机解决此问题，效率比 Jacobi 方法高得多，因为奇-偶法仍是一种同步并行算法。

由上述可见，第一步是将问题化成一个个的方程(子问题)分配给各处理机去计算。Jacobi 方法是每台处理机固定算一个网格点(即解算一个方程)。对奇-偶法来说，由于处理机台数减少了一倍，所以分配给各处理机计算的子问题是两个网格点(二个方程)，一个奇点和一个偶点，如果处理机的台数更少，则分派给每台处理机计算的点更多。这是“分而治之”的原则的应用。每二步是将算法优化时把原来 Jacobi 方法计算未知量的次序改成先计算“偶”点，再计算“奇”点的顺序，以提高算法的收敛速度，也适应处理机的台数。这是“重新编序”原则的应用。

奇-偶法的公式可写成：

$$\begin{cases} u_{\text{偶}}^{(k+1)} = f_1(u_{\text{奇}}^{(k)}), \\ u_{\text{奇}}^{(k+1)} = f_2(f_1(u_{\text{偶}}^{(k+1)})), \end{cases} \quad k=0,1,2,\dots$$

从上式消去  $u_{\text{偶}}^{(k+1)}$ ，则得公式

$$u_{\text{奇}}^{(k+1)} = f_2(f_1(u_{\text{奇}}^{(k+1)})), \quad k=0,1,2,\dots$$

这相当于进行了一次代入消元法，将未知量的个数减少一半，由于  $f_1$  与  $f_2$  都是仿射变换，故  $f_2 \cdot f_1$  仍然是仿射变换。

若方程组为：

$$Au=f$$

经过“重新编序”(相当于行列变换)得到分块形式：

$$\begin{pmatrix} I_{\text{偶}} & -F_1 \\ -F_2 & I_{\text{奇}} \end{pmatrix} \begin{pmatrix} u_{\text{偶}} \\ u_{\text{奇}} \end{pmatrix} = \begin{pmatrix} f_{\text{偶}} \\ f_{\text{奇}} \end{pmatrix}$$

即：

$$\begin{cases} u_{\text{偶}} = F_1 u_{\text{奇}} + f_{\text{偶}} \\ u_{\text{奇}} = F_2 u_{\text{偶}} + f_{\text{奇}} \end{cases}$$

消去  $u_{\text{偶}}$  得：

$$\begin{aligned} u_{\text{奇}} &= F_2(F_1 u_{\text{奇}} + f_{\text{偶}}) + f_{\text{奇}} \\ &= F_2 F_1 u_{\text{奇}} + (f_{\text{奇}} + F_2 f_{\text{偶}}) \end{aligned}$$

若  $F_1$  与  $F_2$  都是压缩算子，“奇”、“偶”各计算一次，相当于“奇”点迭代了两次，故收敛速度加快。

“重新编序”便于构造各种并行算法，由上述讨论看出，如果原来的迭代形式  $u=Fu+f$  是收敛的，则通过左端的未知量代入，消去右端的未知量之后得到的新的迭代形式一般都要比原先的迭代形式收敛快些。其代价可能是使计算公式变得更复杂，由“红-黑”法发展为多色法(multicolor ordering)就是基于这种思想。

如果在网格上按“线”分别标上“○”与“×”(见图 2-2)。若每台处理机计算  $N-1$  个点，使用  $(N-1)/2$  台处理机的 SIMD 计算机，先计算“○”点上之函数值，再计算“×”点上之函数值。若每台处理机在解子问题( $N-1$  阶方程)时用直接法，则计算公式为：

$$\begin{cases} u_{\text{o}}^{(k+1)} = F_1 u_{\text{o}}^{(k)} + f_{\text{o}}, \\ u_{\text{x}}^{(k+1)} = F_2 u_{\text{o}}^{(k+1)} + f_{\text{x}}, \end{cases} \quad k=0,1,2,\dots$$

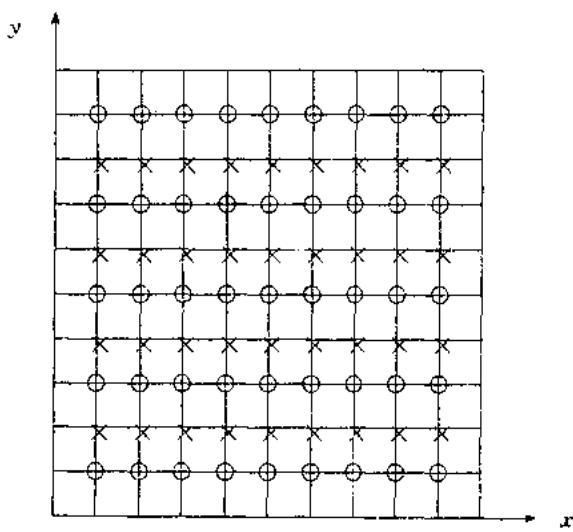


图2-2 红-黑网格图

其中  $u_0$  表示  $u$  在“○”点上的函数值组成的未知向量,  $u_x$  表示在“ $x$ ”点上的函数值组成的未知量, 这是一个同步并行算法。如果引入松弛因子, 则这种算法与带松弛因子的逐线超松弛法(SLOR)收敛一样快。

按“线”的顺序分块的方程, 其系数矩阵是“块三对角”的, 因为每一条线上的未知量仅与它上面的一条线及它下面一条线上的未知量有关。如果采用代入消元法将  $u_0$  消去:

$$\begin{aligned} u_x &= F_2(F_1u_x + f_0) + f_x \\ &= F_2F_1u_x + (f_x + F_2f_0) \end{aligned}$$

则  $F_2F_1$  仍然是“块三对角阵”。这时问题的规模已缩小一半, 即未知量已经少了一半。如果注意到原方程组是块三对角的话, 就会知道这一消去过程是“同步并行”的。在消去了  $u_0$  后, 网格上只留下有“ $\times$ ”的点上的未知量, 如果把剩下这些点又隔一行标上新的 $\otimes$ , 把  $u_x$  又分成了两部分,  $u_0$  与  $u_x$ , 且每一线上的未知量又可以采用直接方法写成(求逆):

$$\begin{cases} u_{\otimes} = G_1u_x + g_{\otimes}, \\ u_x = G_2u_{\otimes} + g_x. \end{cases}$$

若采用代入消元法消去  $u_{\otimes}$ , 则  $u_x = G_2G_1u_x + (g_x + G_2g_{\otimes})$ .

它仍然具有“块三对角”的形式。这一消去过程又构成一个同步并行进程。继续下去, 最后只留下一条线上的未知量( $N-1$ 个), 再须一次便可解出了, 然后逆转回代, 得到全部未知量。

这是一种直接方法, 是解三对角方程组的奇-偶消去法的推广。这种算法反复运用“分而治之”与“重新编序”的原则。

SIMD 计算机的同步是由计算机硬件控制的, 因为它是单指令流, 故在程序设计时, 它使用的语言基本上与常用的串行程序一样。有少数 MIMD 计算机也由硬件控制同步, 成为一类同步 MIMD 计算机, 上面提到的算法自然可以在它上面应用, 有些算法本身就具有很好的并行性, 如共轭梯度法主要是由向量运算组成, 故它只需略加改造就可在 SIMD 机器上使用。

### § 2.3 异步并行算法

对于 MIMD 计算机,这里指的是异步并行的多计算机系统与多处理机系统。因为它是多指令的,故能同时执行两个以上的进程(process)。

MIMD 计算机的一个并行算法是  $k$  个并发进程(concurrent process)的集合,这些进程可以同时地和协作地运行,以解决一个给定的问题。如果  $k=1$ ,则称它是一个串行算法。

为保证一个并行算法能正确而有效地解决所给的问题,进程之间需要交换信息与同步。因此,一个并行程序中可以有一些“相关点”,在这些“点”上进程之间要彼此“通信”。这些相关点把进程分成若干“段”。在每“段”的终点,这个进程可以与其它进程通信,然后再开始计算下一“段”。

一个并行程序中如果这种“相关点”很多,说明进程之间的独立程度很小,每个进程都是由许多小“段”接起来的,这里的“多”与“小”都是一相对的概念,人们用“粒度”(granularity)来表示,由许多许多小“段”组成的进程的小粒度(small granularity)。反之,由运算时间长的“段”组成的进程一定是大粒度的(large granularity)。

粒度小的程序,进程之间的相关性很大,它们要在“相关点”上彼此等待,以期得到对方的信息,则“相关点”变成了“同步点”。在 MIMD 计算机上这种“同步”开销是很大的,同步点很多的算法实际上已失去并行计算的优越性。我们称它为“同步化的并行算法”,以区别前一节讲的“同步并行算法”(synchronous parallel algorithm)。

不需要“同步化”的那些并行算法,则称为“异步并行算法”(asynchronous parallel algorithm)。它的进程之间无需互相等待,进程之间交换信息的方式或是通过动态地读出共享内存中的更新了的数据来实现,或是通过随机地相互使用对方送来的信息,这里说的“动态地”与“随机地”是强调进程既不等待对方更新数据,也不等待对方传送的信息。

下面我们仍用模型问题来说明。

若有  $(N-1)^2$  台处理机组成了 MIMD 系统,我们指定每台处理机计算一个点,

$$P_{i,j}: u_{i,j}^{(k)} = (u_{i+1,j}^{(s)} + u_{i-1,j}^{(t)} + u_{i,j+1}^{(r)} + u_{i,j-1}^{(l)}) / 4 + h^2 f_{i,j} / 4 \\ i, j = 1, 2, \dots, N-1, \quad k = 1, 2, 3, \dots$$

其中  $P_{i,j}$  为处理机台号,  $u_{i,j}$  为该处理机负责计算的近似解在点  $(x_i, y_j)$  处之值,  $u_{i,j}^{(0)}$  为初始猜测, 上标  $s, t, r, l$  表示求得该值的时间。

假定  $u_{i,j}$  ( $i, j = 1, 2, \dots, N-1$ ) 都放在共享内存中,由于处理机的运算速度可以不同(无硬件保证同步),  $P_{i,j}$  在计算  $u_{i,j}^{(k)}$  时用到其它 4 台处理机  $P_{i+1,j}, P_{i-1,j}, P_{i,j+1}$  与  $P_{i,j-1}$  分别计算得到的值,  $u_{i+1,j}^{(s)}, u_{i-1,j}^{(t)}, u_{i,j+1}^{(r)}$  与  $u_{i,j-1}^{(l)}$  的时刻  $s, t, r, l$  可能不相等,于是出现“一片混乱”状态(说不清是用的那一个时刻更新的数据),这种算法是一个“异步并行算法”,因为有  $(N-1)^2$  个进程在同时地、协作地运行,而且进程之间又不需要互相等待(不要“同步”)。

这种异步并行算法也被称为“混乱松弛法”(chaotic relaxation)。已经证明,这种算法是收敛的。

类似地,如果分配每台处理机各负责计算一条线上的  $(N-1)$  个函数值( $N-1$  阶子问题),而且各台处理机所用来求解负责计算的子问题的方法不同,也会引起数据更新与读数的混乱局面。这种混乱计算方法也是一个异步并行计算法。所以在不要“同步”时出现的各

种“动态地”读与“随机地”收、发信息的现象就成了异步算法的主要特征。

前面提到的“分而治之”的原则在异步并行算法中自然已经用到，而“重新编序”则变成了“随机编序”或“混乱编序”与“自由编序”了。

下面讨论解这一类算子方程的异步并行算法(或称异步迭代法)：

考虑乘积空间：

$$E = \prod_{k=1}^n E_k$$

其中每个  $E_k$  是实向量空间，其模记为  $\|\cdot\|_k$ 。每个向量  $u \in E$  记为

$$u = (u_1, u_2, \dots, u_n),$$

其中

$$u_k \in E_k (k=1, 2, \dots, n).$$

定义  $E$  中的模为

$$\|u\| = \max_{k=1}^n \frac{1}{r_k} \|u_k\|_k$$

其中

$$r_k \in R, r_k > 0 (k=1, 2, \dots, n).$$

解迭代形式的方程

$$u = G(u),$$

其中

$$G(u) = (G_1(u), G_2(u), \dots, G_n(u))$$

为算子,  $G: E \rightarrow E$ 。

给定初始向量  $u^0 \in D(G)$ , 依下述迭代公式产生的向量序列  $\{u^p\}_0^\infty, u^p \in E$ ,

$$u_k^{p+1} = \begin{cases} G_k(\dots, u_{j(p)}^j, \dots), & \text{若 } k \in j(p), \\ u_k^p, & \text{若 } k \notin j(p), \end{cases} \quad k=1, 2, \dots, n,$$

其中

$$J = \{J(p)\}_{p \in N}$$

中由集合  $\{1, 2, \dots\}$  的非空子集组成的序列

$$s = \{s_1(p), s_2(p), \dots, s_n(p)\}_{p \in N}$$

是  $N^n$  之元素组成的序列, 它具有如下性质:

I )  $s_k(p) \leq p \forall p \in N$ ,

II )  $\lim s_k(p) = \infty$ ,

III )  $k$  在集合  $J(p)$  中无穷次地出现,  $p = 0, 1, 2, \dots$ 。

如上方式, 由  $(G, u^0, J, s)$  定义了一个异步并行算法。

## 第二部分 并行编程语言

第三章 CM Fortran 概述

第四章 CM Fortran 控制结构

第五章 CM Fortran 数组与数据处理

第六章 CM Fortran 数组变换

### 概要

本部分介绍在 Fortran 90 基础上扩充而成，在 CM (Connection Machine) 计算机上使用的 CM Fortran 编程语言。我们之所以用大量篇幅叙述它，主要出于以下两点考虑：一是这种语言是一种很有前途的并行语言；二是作者使用这种语言编制过大量程序，在 CM-5 计算机上进行计算并获得成功，积累了对这一语言的很多深刻认识。



## 第三章 CM Fortran 概述

CM Fortran 是作为 Fortran 77 扩展 Fortran 90 之数组处理功能的 CM 系统的主要程序设计语言。CM Fortran 支持 Fortran 77 有关控制分配和存取常驻在控制处理机上数据的所有特征。有些限制在用 Fortran 77 特征时被加强,这些特征依赖于常驻于并行处理(分布)存储的数据次序。

以下的 CM Fortran 特征来源于 Fortran 90,并对 Fortran 90 功能有所扩充。

- 阵列赋值
- 阵列部分区段,含由向量值描述的部分区段
- 阵列构造
- 整个阵列或阵列部分的算术和逻辑运算
- 有关阵列或标量基本的内部函数
- 阵列转换内部函数
- 内部函数参量关键词
- 带有 ALLOCATE 语句的二进阵列分配
- 阵列指针(非 TARGET 分布或指针赋值语句)
- CASE 结构
- WHERE 语句和块 WHERE 结构
- 数组值外部函数
- 指定外部子程序和函数的接口块

CM Fortran 也包含 Fortran 90 的下列特色;这些特色形成部分是 DOD 军用标准的推广(MIL-STD-1753)。

- 位——处理静态函数
- DO WHILE 语句
- END DO 语句
- INCLUDE 行
- 在 IMPLICIT 语句中 NONE 说明符

最后,CM Fortran 包括的阵列功能来源于早期的草案,它是对 Fortran 90 标准的扩充。它包含:

- DO TIMES 语句
- FORALL 基本阵列赋值
- 内部函数 DIAGONAL, RANK, REPLICATE, PROJECT, FIRSTLOC 以及 LASTLOC。

### § 3.1 CM Fortran 的模式

CM Fortran 程序包含几种不同的源构造,编译器将其转化为两种编码束:串行码和并行码。串行码在控制处理器上(又称前端处理机)执行,并行码在并行处理器上执行。如图 3-1 所示。

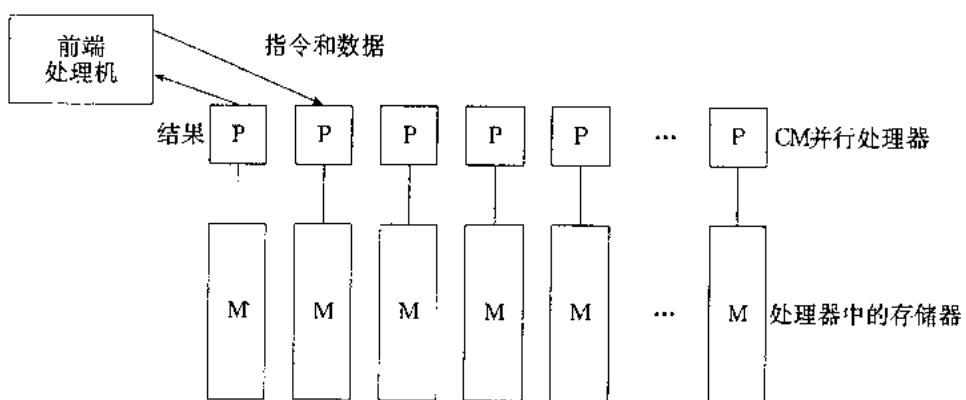


图3-1 CM机的执行模式

前端处理机(简称前端机)是 CM 并行处理器(简称 CM 机)系统的用户入口。它提供软件开发和调试工具以及用户熟悉的程序运行环境。从用户的观点看,CM 机的环境可看作是通常前端机环境的扩展形式。另外,前端机还提供一组常用工具和语言。环境包括至少一组 CM 机语言的常驻汇编解释程序。前端机还有称为前端总线接口(FEBI)的专用硬件,以实现跟 CM 机并行处理部件通信。

前端机不仅担任标量数据的运算,而且为系统提供一个编程环境,程序可以放在前端机的磁盘上。前端机还控制整个数据并行程序的执行,涉及并行数据的程序通过一个接口被送到并行处理单元中,在那里瞬间便被扩散到所有的处理器去执行。并行数据的操作运算通过前端机向 CM 并行处理器所发的命令来实现。这种工作方式的好处是:首先程序员可以在他熟悉的环境中工作。由于前端机和 CM 并行处理器是相互作用,因此它的编程语言、调试环境和前端处理机的操作系统都保持相对不变。第二,对许多应用程序来说,很大部分程序并不是计算的,而是用来处理程序、操作系统和用户之间的界面上的问题;如接收键盘输入,解释用户确定的参数,在终端上显示信息等等。由于程序控制保留在前端机上,因此已经为前端机所开发的那些程序,无论有没有 CM 并行处理单元,都仍然可用,只有那些专门附属于驻留在 CM 处理器中数据的程序才需要用到数据并行语言扩展。其次,一部分程序不是严格属于计算性质的,可以在前端机上执行,而另一部分以并行方式运行则是十分有效的程序,即对数据集里的并行数据部分进行运算,由 CM 处理器来运行。这样串行的前端机和并行的 CM 处理器两者的优势都被充分的发挥出来了。

在串行码中包含影响一个程序的执行控制流的语句,比如 IF,DO 和 CALL 语句,也包含影响在控制处理存贮中仅有数据的操作。标量数据由前端处理机处理,依赖于编译时执行模式,控制处理器可以是分划管理者或 CM-5 的单个结点或 CM-2 或 CM-200 的前端处

理机(有关各种执行模型请参看本书提供的有关文献)。

受数组运行制约的数组被分配到并行单元的分布存储之中,数组运算由并行处理器执行。由于历史原因,CM Fortran 称分布数组为 CM 数组。所谓并行处理器和处理单元指的是执行并行运算的 CM 系统的组成成分,取决于所选用的执行模式,它们可能是结点或 CM-5 的向量单元,或 CM-2 或 CM-200 的结点。

CM 系统软件在每个并行处理器的存储器内动态地分配一个或多个数组元素,因此使得应用编码独立于执行机器的物理处理器数目。这种码的范围从一种机器到另一种机器,无需重新编码或重新编译。

在数组元素间操作或通信的最快途径是每个并行处理器内的局部存储存取。对数组格式的运用(用编译指定 LAYOUT)能最大限度地利用这种途径。

为了在处理机间转换数据,CM 系统软件支持三种处理机间通信途径,执行为运行时间通信子程序。编译器设法寻求适当的途径转换给用户。

**一般通信:**最普通的途径允许数据从任意处理机直接送到任意别的处理机,多台处理器能同时传送数据给其他处理器。这种途径在操作中是经济的。

**网络通信:**一种较为有效些的途径把数据视为  $n$  维网,并且允许每个处理器把数据立即传送到网中邻近的处理器上。这种处理器间的通信,用一种专门的高速网络来实现。那些相关的数据单元所在的处理器会存放一个彼此间相互指向的指针。当数据需要调用时,它将通过网络到适当的处理器中去调用。该网络支持通用的通信方式,附加的特殊硬件支持某些规则通信方式。在一个多维的矩形网络上,最邻近通信是特别有效的一种通信方式。

**全局通信:**第三种途径累积地在一个数组或按一维方式执行操作,该累积操作可使一个数组或一维向量成为标量结果。换句话说,它可以遍历一维向量而将一个一维向量元素积累为一组结果。

CM Fortran 中数组赋值常常不需要任何处理机间通信,但它在某种情况下可以要求网络通信或普通通信。在数组段,向量值下标要求普通通信途径,即处理像 PACK(压缩函数)和 UNPACK(展开函数)这种内部转移函数。在操作常规的数组维数的转移时,内部函数 CSHIFT(循环移位函数)通常用网络通信。

## § 3.2 CM Fortran 的结构与特点

### 1. 并行数据结构的建立

在 CM Fortran 中表达数据并行的程序可以用串行程序中相同的数据来描述,重点是用数组这样的一致数据结构,其中的元素能同时进行处理。例如:赋值语句  $A = B + C$ ,在串行语句中就是单个数  $B$  与  $C$  相加,结果存入  $A$  中。在并行语言中,如果  $A, B, C$  说明规模大小为  $n$  的数组,那么这个语句就等效地表示了  $n$  个同时进行的加法操作。在这种情况下,每个数组元素的数量超过物理的硬件处理器的数量,一种虚拟处理器就发挥作用,对用户而言就好像有无限多个处理器一样。在这里  $A, B, C$  可以是标量,也可以是向量,矩阵或多维数组。下面是 Fortran 77 与 CM Fortran 的程序段的对照表示。

Fortran 77		CM Fortran
DO 10 I=1,50	}	
DO 20 J=1,100		A = B + C
A(I,J)=B(I,J)+C(I,J)	}	
20 CONTINUE		A(:,2:100:2)=B(:,2:100:2)+
10 CONTINUE	}	
DO 30 I=1,50		C(:,2:100:2)
DO 40 J=2,100,2	}	
A(I,J)=B(I,J)+C(I,J)		A(:,2:100:2)=B(:,2:100:2)+
40 CONTINUE	}	
30 CONTINUE		C(:,2:100:2)

在 CM Fortran 中, 编译程序会自动根据这个数据如何用于该程序来决定这个数组究竟是作并行处理, 还是作串行处理, 是作单独的加法操作, 还是作几千个加法操作以及如何将并行数据分配(映射)到处理器上去。程序中也可以插入结构化的注释说明来驾驭这些自动作出的判断。

## 2. CM Fortran 与当前结构程序设计方式的兼容性

数据并行程序设计很容易用于大规模并行计算机, 也适合于串行机、向量机以及共享主存的并行计算机。

传统结构化程序设计中, 最简单的赋值语句, 最常用的基本控制流用 BEGIN…END, 条件转移用 IF…THEN…ELSE, 循环用 WHILE…DO 描述。同样, 在 CM Fortran 中的并行程序设计中, 最简单的是局部计算, 最常用的是机器间通信, 它可以用重复, 归约, 置换和并行前缀来描述。现对比表示如下:

结构程序与并行程序的对比

	结构程序	并行程序
基本计算	赋值语句	局部计算
基本模式	BEGIN…END IF…THEN…ELSE WHILE…END	重复, 归约 置换, 并行前缀
公共组合模块	DO 循环 CASE 语句 REPEAT…UNTIL	规则网格 移动, 取和, 排序, 快速变换
转移	GOTO	信息传递

例如, 假定 P 是一个长度为 N, 由整数 1 到 N 的向量。那么 P 是一个置换向量, 语句 A = A(P) 是对数组 A 进行这一置换(语句 A(P)=A 则是进行逆置换)。

又如: 在数组变换中, 内部函数 CSHIFT 是对数组进行循环移动。若有语句

$B=CSHIFT(A,2,1)$

是把数组 A 向左移一个位置(即沿第二维方向移一个位置),并把结果存于数组 B 中。

归约函数用于数组的并行前缀操作,从而得到一个新数组,该新数组的每个元素是旧数组中该元素之前(含该元素)所有元素归约的结果。例如下面的前缀加法使得 B 中的每个元素是 A 的相应元素之前的所有元素和:

$B(I)=SUM(A(1:I))$

即:

A(I)	1	3	5	7	9	11	13	15	17	19
↓										
B(I)	1	4	9	16	25	36	49	64	81	100

这说明传统结构化程序设计能表达的语义,并行程序设计同样能表达。

### 3. 数组操作

在 CM Fortran 中,将算术、逻辑和字符等操作以及内部函数全部扩充为可以对值数组的操作。这些操作包括对整个数组、子数组(即数组段或部分数组)和屏蔽数组的赋值,值数组的常量、表达式和函数(内部函数和外部函数)在表达式中可同时含有多个数组或数组段,但它们的大小必须一致。标量也能自由地混杂到数组操作中,实际操作时,完成并行计算的各处理机上都预先得到该标量的一个副本。例如,假定 A,B 是  $10 \times 10$  的矩阵,C 是  $10 \times 50 \times 100$  的数组,E 是一个标量,那么就可以进行如下运算:

$A=B * 8 + C(:,1:10,9) + E$

此处表达式中的 8 和 E 也可以理解为两个  $10 \times 10$  矩阵且元素值分别是 8 和标量 E。因为标量在数组值的表达式中,它们会被自动加以复制和表达式中的其它数组一致起来。

数组操作的另一种形式是对 Fortran 中的内部函数的扩充,使其参数既可以是标量也可以是数组。例如:若 X 和 Y 是  $10 \times 10$  的数组,则语句  $Y=SIN(X)$  将会使 100 个处理机同时作 SIN 函数的运算。

在 CM Fortran 中,提供了丰富而且功能很强的数组运算函数。他们是:数组移动函数,数组归约函数,数组构造函数和高级运算函数。其中,数组移动函数是对数组元素进行重新定位,它包括 CSHIFT(循环移位)函数、EOSHIFT(截止移位)函数和 TRANSPOSE(矩阵转置)函数等。数组归约是将一个旧数组进行归约处理后得到一个新数组,它包括数值函数和逻辑函数:数值函数有 MAXVAL、MINVAL、SUM、PRODUCT;逻辑函数有 ANY、ALL 和 COUNT 等。数组构造函数是根据现有数组元素,按照指定的方式构造一个新的数组。它包括 DIAGONAL(对角线)函数,MERGE(合并)函数,PACK(压缩)函数,UNPACK(扩散)函数,REPLICATE(复制)函数,SPREAD(扩展)函数和 RESHAPE(重新整形)函数等。高级运算函数有 DOTPRODUCT(向量点积)函数和 MATMUL(矩阵乘法)函数。这些数组变换的内容,将在第六章中详细介绍。

### 4. 动态数组

在 Fortran 77 中,对于大小需求不明确的数组,在程序中常常先估计说明一个比通常需

要和规模大很多的数组,然后再按实际数据进行输入。这种方法使用数组,要么只使用了数组的一部分,要么超过了数组说明时规定的大小而产生错误。带来的问题是,要么浪费计算机内存,要么仍然解决不了实际问题。而 CM Fortran 提供了动态分配数组功能。

动态数组的维数上下界是在程序的执行过程中随时按需要变化,数组需要占多少内存,就可以在程序的运行过程中动态地分配给数组多少内存。如果该数组以后不再使用,又可释放该数组,把该数组占用的内存归还给系统另作它用。这样既节约内存使用,又提高了有限的内存的使用效率。实现数组的动态分配的语句是 ALLOCATE 语句;释放数组存储空间的语句是 DEALLOCATE 语句。动态数组的实现程序举例如下:

```

SUBROUTINE SUBname(a,b,c,d)
! a,b,c,d 的大小仅在运行时刻才确定的数组
REAL,ALLOCATABLE DIMENSION(:,:);:a,b,c,d
! 说明 a,b,c,d 均为实型的二维动态数组
INTEGER::n,m
...
READ *,n,M
! 子程序运行到此处,需要建立数组 a,b,c,d 的大小
ALLOCATE(a(5:n,10:m),b(n,m),c(m,m),d(n,n))
! 使动态数组 a,b,c,d 分配所需数量的内存空间
...
DEALLOCATE(a,b,c,d)
! 将动态分配给数组 a,b,c,d 的内存释放
...
END SUBROUTINE SUBname

```

### 5. 并行控制语句

并行控制语句可以控制不同处理器在同时刻进行不同操作,也可以对部分处理器给予限制,并行控制语句中的 WHERE 语句,是用一个逻辑数组作为对一个一致的数组表达式中元素的屏蔽操作。

例如:语句 WHERE(A.GE.0)A=SQRT(A)是用平方根来替换数组 A 中的大于等于 0 的元素,负的元素不变。又如假定 A,B 分别是  $10 \times 10$  的矩阵,则 WHERE(B.NE.0)A=A/B 等价于:

```

Loop_I: DO I=1,10
Loop_J: DO J=1,10
        IF(B(i,j).NE.0) A(i,j)=A(i,j)/B(i,j)
        END DO Loop_J
    END DO Loop_I

```

WHERE 语句的另一种形式是 WHERE 结构,它可以执行两个以上数组的分配,并且还可以确定对那些测试失败的元素进行分配。例如:

```

WHERE(B.NE.0)
C=A/B

```

```
ELSEWHERE
```

```
C=1.0E 30
```

```
END WHERE
```

这里,A,B,C 都是一致的数组,在含有 B 的非零元素的每个处理器里把 A/B 的结构分配到 C,同时在包含 B 的零元素的每个处理器里把 1.0E 30 这个值分配给 C。

FORALL 语句是另一种并行控制语句,FORALL 语句可以用索引表达式来选择激活的处理器而不是用屏蔽,或者除了屏障外还可以用索引。

例如:对  $n \times n$  阶矩阵 A 进行初始化:

```
FORALL (I=1:N,J=1:N)A(I,J)=1.0/REAL(I+J-1)
```

在这里,FORALL 语句中对 A 的赋值是并行地执行的。

又如:用 FORALL 语句屏蔽矩阵 A 中的部分元素:

```
FORALL (I=1:N,J=1:N,I.NE.J)A(I,J)=0
```

结果是正方形矩阵 A 中除对角线的元素外,其余元素值均为 0。

## 6. 指针

指针允许动态地测定数组大小、排列数、链接诸结构,并建立链表、树、图。任一内部类型或派生类型的对象均可被说明具有指针属性。

指针不是数据实体,而是某种实体的属性,它必须借助一种形式新颖的说明来定义。例如下列语句:

```
TYPE (NODE),POINTER::FIRST,NEXT,TMP
```

说明了三个指针对象;FIRST,NEXT 和 TMP,它们均指向派生类型 NODE 的对象。又如:

```
REAL,DIMENSION(:,:),POINTER::IN,OUT
```

说明了两个指针对象 IN 和 OUT,它们都指向两维实型数组。

## § 3.3 输入输出初步

所谓输入是指数据从外部介质(如键盘、磁带、磁盘等)传输到计算机内存(例如从磁盘上将数据文件中的数据传输到内存)的过程,称为输入或读入数据。而数据从内存传输到外部介质(如打印机、显示器、磁带、磁盘等),例如将计算机内存中的数据送到打印机上打印的过程,称为输出或写出数据。用于数据输入/输出的语句称为读语句(READ 语句)/写语句(WRITE 语)或打印语句(PRINT 语句)。

### § 3.3.1 输入语句(READ 语句)

#### 1. READ 语句的最简格式

READ 语句是 CM Fortran 实现输入数据的主要手段。而 READ 语句最简单的输入形式为表控输入,其格式为:

```
READ *,变量表
```

其中:“\*”表示使用系统隐含的表控格式读入数据;

“变量表”是指所使用的变量名表,它由一个或多个用逗号分隔的变量名(或者是数组名,数组片段名,结构成员名等)所组成。

所谓“表控输入”是指由计算机系统所指定的设备将数据输入到变量表所指定的变量中去。这里所说的“计算机系统所指定的设备”是指计算机系统所规定的某些外部设备，这些设备专门用于数据的表控输入，如键盘等。在程序设计时，程序员不必专门指定这些设备就可直接实现数据的输入。

例如： READ \*,x,y,z

这个输入语句的输入变量列表由 x,y,z 组成。这条语句的执行，就是从系统所指定的输入设备上，读入三个数据值，分别赋给变量列表中所列的变量 x,y,z。若输入设备指定为键盘输入，则只要从键盘上输入三个数值并回车，就完成了该语句的执行。若 x,y,z 说明为整形，当程序执行到此语句时，输入如下数据：

18,19,25↙或 18-19-25↙

则 x 得值 18， y 得值 19， z 得值 25。

若 x,y,z 说明为实型，则输入为：

18.0,19.0,25.0↙

使用表控格式时，输入数据的类型必须与输入表中变量说明的类型一致，否则出错。

## 2. READ 语句中带部件号的格式

使用表控格式的读语句一般形式为：

READ(部件号, \*)变量名表

其中：“部件号”是用以指明输入/输出是在哪个部件上进行的。与计算机相连接的读写设备很多，必须在读写语句中注明在哪个设备上读写，使机器有所遵循。在 CM Fortran 中，读写语句的设备用一个正整数代表，这个号码称部件号，它是某台外部设备的代号（例如终端的部件号为 5）。至于每个设备的部件号具体是何值，则根据使用的计算机系统不同而异。因此，使用前应查阅有关资料。

“\*”表示在输入数据时，是按系统隐含的表控格式输入。

显然，当部件号代表当前正在使用的设备时，下面三个读语句是等价的：

READ(5, \*)变量名表

READ(\*, \*)变量名表

READ \*, 变量名表

它们都表示在当前设备上，按表控格式输入若干数据赋给输入表中各变量。

## 3. READ 语句中带控制信息表的格式

在输入/输出语句中，除了三个不可缺少的条件（输入/输出设备，输入/输出的格式，输入/输出的变量）外，还可以有许多附加功能，例如指出读入数据出错时将如何处理错误（即增强程序的容错能力）等。这些功能每一个可写成一个控制说明符，由程序员按需要选用，所有被选用的说明符彼此间用逗号分开，全部控制说明符用一对括号括起，称为控制信息表。带控制信息表的 READ 语句的一般形式为：

READ(说明符 1, 说明符 2, …, 说明符 n)变量名表

其中：“说明符”的一般格式为：关键字=指定参数。

在控制信息表中，常用的说明符功能有下列形式：

READ(UNIT=部件号, FMT=格式说明符, IOSTAT=整型变量)变量名表

其中：“UNIT=部件号”是部件说明符，用来指定输入设备，关键字是 UNIT，在“=”号后

填写一个整数值,指定输入的部件号;

“FMT=格式说明符”是格式编辑符,用来指定编辑格式,关键字是 FMT,在“=”号后填写数据输入时规定的格式;

“IOSTAT=整型变量”是输入输出状态说明符,用来指明输入输出时的状态,关键字是 IOSTAT,在“=”号后是一个整型变量,当程序执行了此 READ 语句后,变量将获得不同的值,该值指出读语句所处的状态;

$$\text{整型变量值} = \begin{cases} 0 & \text{读入正常} \\ \text{正整数} & \text{读入出错} \\ \text{负整数} & \text{遇文件结束} \end{cases}$$

需要注意的是,输入输出状态值的正值与负值对不同的机型有不同的规定,请按有关机器的说明资料执行。

例如下列 READ 语句都是正确的:

READ(UNIT=5,FMT=(F8.6),IOSTAT=K) omg

或                   READ(5,'(F8.6)',IOSTAT=K) omg

在后一个 READ 语句中省略了“UNIT=”和“FMT=”等关键字,只写部件号及格式说明是允许的。

### § 3.3.2 输出语句(PRINT 语句和 WRITE 语句)

在 CM Fortran 中,PRINT 语句是实现输出的最简方法,其输出形式为:

PRINT \*,输出列表

其中:“\*”表示为表控输出;“输出列表”是由一个或多个逗号分隔开的变量名(或数组名、数组片段名、结构成员名、字符串名等),常数,表达式组成(在读句的输入表中,则不允许出现常数和表达式)。

同表控输入一样,表控输出无需程序员指定输出设备。输出设备由计算机系统指定,通常为终端显示器或打印机。例如:

该语句表示在部件号 6 的设备上按每个数据占 10 列, 小数部分占 5 列的格式在一行内打印出 X,Y,Z 三个值, 当输出正常时, 机器给 K 置零值, 输出错误时, 机器给 K 置正整数值。

### § 3.4 格式语句

在 CM Fortran 中, 输入/输出语句中的自定格式不仅可以由控制信息表来实现, 而且也可以像 Fortran 77 一样, 由专门格式语句(FORMAT 语句)来实现自定格式的输入/输出。这种多种方法的输入/输出形式, 为程序员编写各种形式的控制表格提供了极大的方便。

#### 1. FORMAT 语句

FORMAT 语句用来指明输入/输出的格式, 它描述了数据以什么样的格式进行内外形式的转换和编辑方式。FORMAT 语句可以出现在程序的 PROGRAM 语句之后, END 语句之前的位置。

FORMAT 语句是一种非执行语句, 它必须有标号。一个输入/输出语句要使用一某个指定的格式, 则这个 FORMAT 语句的标号一定要出现在该输入/输出语句的格式说明符中。

FORMAT 语句的一般形式为: f FORMAT(s)

其中:f 为语句标号;s 为格式说明

格式说明 s 是编辑描述符的项目表, 即可使用在 FORMAT 语句中, 也可以以字符串的形式'(s)'直接出现在读/写语句中。例如:

```
      WRITE(*,100)A,X,Y
      100 FORMAT(2X,I6,3X,F10.3,3X,F10.3)
```

在上述的 FORMAT 语句中的包含的编辑描述符有:

F: 实型编辑描述符(F 型)

X: 空格编辑描述符(X 型)

#### 2. 编辑描述符

编辑描述符用来指定数据的输入/输出格式, 通常编辑描述符有下列一般形式:

rIw 整型编辑描述符(I 型)

rFw.d 实型编辑描述符(F 型)

rEw.d 实型的指类型编辑描述符(E 型)

rAw 字符型编辑描述符(A 型)

nX 空格型编辑描述符(X 型)

nHhIhzh...hn 文字型编辑描述符(H 型)

'撇号编辑符

rLW 逻辑型描述编辑符(L 型)

其中:r 是非零无符号整常数, 称为重复说明, 表示一个描述符连续重复出现的次数;

w 是非零无符号整常数, 表示字段的宽度;

d 是无符号整常数, 表示实数中小数部分的字段宽度。

在这里我们仅就最常用的 I,F,X 编辑描述符作一概要的说明:

(1) I 型编辑描述符(Iw)

整型编辑描述符用来控制整型数据的格式变换。

在输入时,整型编辑符 Iw 的功能是从外部介质上读入一个数并转换成整型的内部形式后赋给输入表中相应的元素。

在输出时,将输出表中相应的整型元素传送到 w 个字符宽的外部字段上,并在 W 范围内向右对齐,负号作为最左非空字符。

#### (2)F 型编辑描述符(Fw.d)

F 编辑描述符说明实型或双精度数据的转换和编辑,或者说明复型数据的实部或虚部的转换和编辑。

在输入时,从外部介质上读入一个数据转换成实型或双精度型的内部形式赋给输入表中相应的元素。在输出时,将相应的输出表中的元素的值,进行四舍五入,精确到小数点后 d 位,并传输到 w 字段宽的外部字段上,在 w 范围内向右对齐,实现定点形式输出。

#### (3)X 型编辑描述符

nX 编辑描述符表示在输入时跳过 n 个字符,在输出时留下 n 个空白,即把下一个将要输入/输出的字符位置向前推移了 n 列。

nX 编辑描述符可以避免相邻的两个数据紧连在一起,使下一个数据项向前推移 n 列。特别是在输出时,可使输出结果相互分开,排列整齐,容易阅读。

## 第四章 CM Fortran 控制结构

在 CM Fortran 中,按其程序结构可分为三种:即顺序结构,分支结构和循环结构。如在前面提到的赋值语句、READ 和 WRITE 等语句所构成的程序,都是按其书写的顺序从上到下,自左至右依次执行,故称此类程序为顺序结构。在本章中将引出的条件语句,重复语句所构成的程序结构则分别称为条件控制结构和循环控制结构。在条件结构中控制条件的执行是由条件语句来决定的,而并行程序中执行控制条件是限制一些运算操作对某个数据结构的数据单元子集的作用。条件操作首先要对并行数据结构的所有单元里的规定条件进行测试,然后仅对那些条件为真的数组单元执行操作。和串行程序一样,并行程序中的条件语句也可以进行嵌套。

在循环控制结构中,根据循环条件,若循环条件成立,则并行执行循环体内的各语句。此时各个独立的处理器执行循环体内的语句若干次,每个处理器可执行不同的次数,这取决于被处理的数据。如果在循环期间,处理器并不相互作用,则每个处理器各自重复执行适当次数。当所有的处理器都完成了所要重复的次数,则所有的处理器又成为同步退出循环。如果循环期间处理器有相互作用时,则由通信语句来完成此功能。

### § 4.1 条件结构

计算机在执行程序时,一般按照程序中语句的先后次序逐句执行的。但是,对于一个稍微复杂一点的实际问题,常常需要依据某些条件来改变执行顺序,选择所要执行的语句,如计算变量  $x$  绝对值的公式如下:

$$|x| = \begin{cases} x & \text{当 } x \geq 0 \\ -x & \text{当 } x < 0 \end{cases}$$

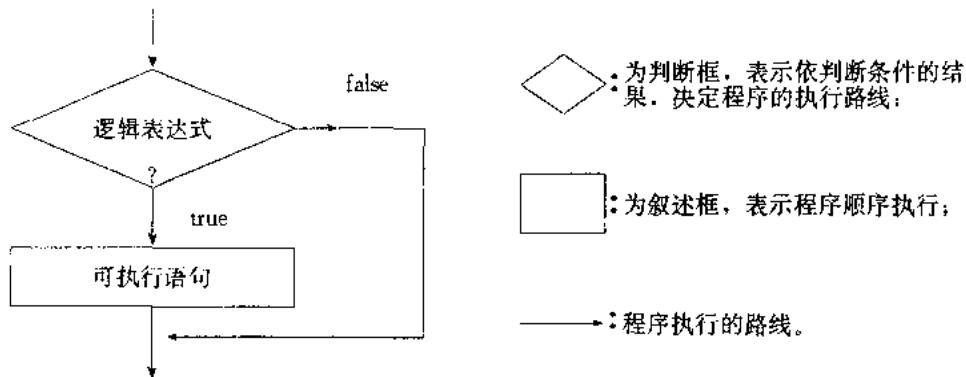
这就是说,计算公式要求根据条件  $x \geq 0$  或  $x < 0$  去执行不同的语句。对于这种有选择地去执行的分支语句,CM Fortran 提供了如下条件语句:

#### 1. 单分支条件语句

单分支条件语句的一般形式为: IF (e) s

IF 是关键字,e 是一个逻辑表达式,指明了一个条件,s 是一个可执行语句。这个语句的功能是:如果条件成立(当逻辑表达式结构为“真”),则执行 s 语句,而后执行下一条语句;如果条件不成立(即逻辑表达式结构为“假”),则不执行 s 语句,此时整个 IF 语句不作任何操作,而顺序执行下一语句。单分支的条件语句的执行过程可用程序流程图描述为图 4-1 形式:

**例 4.1** 乘坐武汉至北京的民航班机,每张客票可免费托运小于等于 20 公斤的行李,大于 20 公斤时,超重部分按每公斤 0.5 元收费,试编制计算收费的程序。



解：可按下列步骤来进行设计：

(1) 变量说明：

$x$ ：表示重量(单位公斤)，是一实数型单精度类型；

$y$ ：表示收费(单位元)，是一实型单精度类型。

(2) 计算式：

$$y = \begin{cases} 0 & x \leq 20 \\ (x - 20) \times 0.5 & x > 20 \end{cases}$$

(3) 写出程序，在这里仅对单分支条件语句，写出计算标准核算收费程序部分：

```
.....
y=0
IF x>20 y=(x-20)*0.5
....
```

在这里， $y$  赋初值 0，当  $x > 20$  时，条件语句执行  $y = (x - 20) * 0.5$  部分，因而更改了  $y$  的初值，当  $x \leq 20$  公斤时，条件语句不执行  $y = (x - 20) * 0.5$  的语句部分，因而  $y$  保持初值不变。

单分支条件语句还可以用来设置对程序出错进行处理的功能。如在设计一个实用的程序时，不仅要考虑程序的通用性、方便性和可移植性等因素，还要考虑系统的容错能力。不能因为运行环境、外部条件，甚至用户操作失误等因素引起的错误而使系统停止运行。因此，在程序设计中，不仅要保证程序的正确无误，能完成预定的功能，而且还要考虑程序能防止和排除各种外部原因引起的错误，不让这些错误影响程序的正常运行。利用单分支条件语句设计一个错误处理程序，采用跟踪技术，一旦发生错误，程序就能及时捕捉错误，对错误进行处理。

IF 语句为：

```
IF (ERR) GOTO 999
```

999 为语句的标号，假定 999 处为错误处理程序段，当程序出现错误时，立即转移到 999 语句处进行出错处理。

单分支条件语句常常用来处理迭代问题的退出，具有独特的效果。在计算机程序中经常作连加、连乘或反复执行某段程序，并规定只有当某个参数大于或小于某值时才停止迭代，这时就需使用单分支条件语句。如有些级数求和时，把精度作为 IF 语句中的  $\epsilon$ ，决定它是否

满足精度而停止求和。例如设无穷级数：

$$y = 1 + \frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^n} + \cdots$$

求  $y$  的近似值, 当某一项的值  $T_n \leq 10^{-8}$  时认为已达到精度要求, 不再继续求和。那么必须在求和的循环体中加入下面的语句：

```
y=y+T
IF (T<1.0E-8) EXIT
```

前一句是每次加入新的一项  $T$ , 后一语句规定当满足精度时, 就退出重复结构, 停止求和(EXIT 是停止循环的执行, 强迫退出本层循环结构体)。

## 2. IF 语句结构块

在日常生活中, 我们经常会遇到多分支的算法, 或者尽管是一个二路分支选择, 但每种选择要执行不止一个语句。为适应这种情况的需要, CM Fortran 提供了多种选择结构块满足不同的需要。

### (1) 最简选择结构块

最简选择结构块的一般形式：

```
IF (e) THEN
    语句块
END IF
```

最简选择块的形式与单分支条件语句相似, 但不同的是语句块中可以包括任意多个语句, 当逻辑表达式  $e$  的值为“真”时, 顺序执行语句块, 然后由 END IF 退出选择块; 当逻辑表达式  $e$  的值为“假”时, 跳过语句块, 执行 END IF 的下一个语句。

### (2) 条件选择结构块的一般形式

IF 选择块的一般形式为：

```
IF (e1) THEN
    语句块 1
ELSE IF (e2) THEN
    语句块 2
ELSE IF (e3) THEN
    语句块 3
...
ELSE IF (en) THEN
    语句块 n
ELSE
    语句块 n+1
END IF
```

其中  $e_1, e_2, \dots, e_n$  是逻辑表达式, 指出各种条件; 语句块  $i$  是一组语句, 是当条件  $e_i$  成立(即  $e_i$  逻辑表达式值为真)时所执行的语句块。

IF 结构的控制可用图 4-2 表示。

控制从人口进入, 经下列步骤:

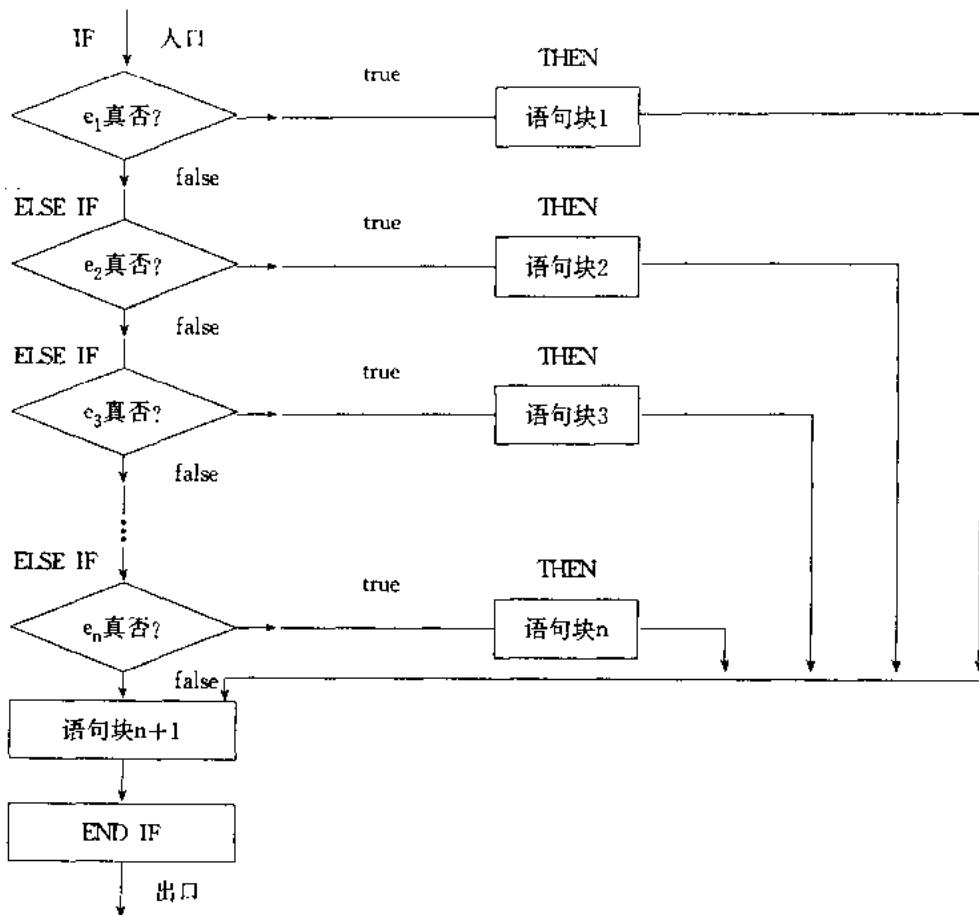


图 4-2

- ① 检查  $e_1$  真否? 若 true 执行语句块 1, 绕过其他块, 直接转出口语句 END IF 处出口; 若 false, 跳过语句块 1 检查  $e_2$ ;
- ② 若  $e_2$  为 true, 执行语句块 2, 而后直接转出口; 若 false, 跳过语句块 2, 检查  $e_3$ ;
- ③ 若  $e_3$  为 true, 执行语句块 3, 然后转出口; 否则检查  $e_4$ , 以此类推;
- ④ 如果所有 ELSE-IF 语句的  $e_i$  都为 false, 那么必须执行 ELSE 语句下一个语句块  $n+1$ 。

从上述控制过程可见, 执行 IF 选择结构, 自上而下顺次检查每块前面的条件, 满足条件的就执行该条件下面的语句块, 执行完该语句块后, 立即转向出口, 以后即使还有语句块满足条件, 也不予理睬, 因此, 一个 IF 结构中最多只执行一个语句块, 也可能什么也不执行, 起空滑作用。

现举例说明 IF 选择结构的应用。

**例 4.2** 按税务部门核收个人所得税规定, 当综合收入(工薪、承包、劳务、租赁四项和)小于 800 元时, 国家免收税款。若在 800 元至 1000 元之间时, 其超过 800 元部分税率为 10%, 1000 元以上的除抽 10% 外, 其超过 1000 元部分再核收 20% 税, 外加手续费 80 元。试编制计算税收的程序。

解：设  $x$  为个人收入总额，单位：元。

$y$  为税收额，单位：元。

则有关系式：

$$y = \begin{cases} 0 & x \leq 800 \\ (x - 800) \times 0.1 & 800 < x \leq 1000 \\ (x - 800) \times 0.1 + (x - 1000) \times 0.2 + 80 & x > 1000 \end{cases}$$

程序可为：

```

PROGRAM TAX
  IMPLICIT NONE
  REAL :: x, y
  y = 0
  PRINT *, 'Input your income:'
  READ *, x
  IF (x > 800) AND (x <= 1000) THEN
    y = (x - 800) * 0.1
  ELSE IF x > 1000 THEN
    y = (x - 800) * 0.1 + (x - 1000) * 0.2 + 80
  ELSE
    PRINT *, 'Exempt from taxation'
  END IF
  PRINT *, 'The amount of tax y = ', y
END PROGRAM TAX

```

### (3) 条件选择结构块的取名

IF 结构可分为无名与有名两种情况，前面讲到的一般形态是无名选择结构，当 IF 选择结构太多或彼此嵌套，为了阅读时清晰，可以给它取名。IF 选择结构的取名规则同 CM Fortran 变量名相同，要尽量反映本结构的算法。条件选择结构的取名形式一般为：

```

条件结构名: IF (e1) THEN
  语句块 1
  ELSE IF (e2) THEN[条件结构名]
  语句块 2
  ...
  ELSE[条件结构名]
  语句块 n+1
END IF 条件结构名

```

若设条件结构名为 block-name，则入口语句为：

```
block-name: IF (e1) THEN
```

那么出口语句必须为：

```
END IF block-name
```

当入口语句写上条件结构名时，出口语句则必须写上条件结构名，并且名字要一致。条

件结构名与入口语句间必须用冒号“:”分隔,而出口语句与条件结构名之间只空一格,不能带冒号“:”。当条件结构有名时,结构中的 ELSE IF 语句以及 ELSE 语句后可以写上结构名如:

```
ELSE IF (e1) THEN block-name
...
ELSE block-name
```

也可以不写结构名,与无结构名时的写法一样。下面是一个带结构名的条件结构块的程序段:

```
OUTER:IF(A.GT.0)THEN
    B=C/A
    ELSE IF(C.GT.0)THEN OUTER
    B=A/C
    D=-1
    ELSE OUTER
    B=ABS(MAX(A,C))
    D=0
    END IF OUTER
```

#### (4) 条件选择结构块的嵌套

条件选择结构中的任何一语句块(IF—THEN 块,ELSE—IF 块,ELSE 块)都可以嵌入另一个结构,被嵌入的结构可以是另一个 IF 结构,也可以是另一些形态、功能不同的结构,如后面将要介绍的 CASE 结构,DO 结构。嵌入的前提是必须把整个结构完整地嵌入 IF 的某一块中,不允许一部分嵌在一个块中,而另部分嵌在另一块中,即为:被嵌入的任何结构不可跨越两块。

为了层次结构清晰,查错及修改方便,可采取缩格书写形式,以区分内层与外层结构。若对内层、外层结构分别取名,那将更加清晰易读。

条件选择结构块嵌套的示意形式为:

```
first:IF (e1) THEN
    语句块 1
second:IF (e11) THEN
    语句块 11
    ELSE
        语句块 12
    END IF second
    ELSE IF (e2) THEN
        语句块 2
    ELSE
        语句块 3
    END IF first
```

**例 4.3 编程求解一元二次方程  $Ax^2+Bx+C=0$  的根。**

解：设  $A, B, C$  分别为一元二次方程的系数； $D$  为根的判别式。

则根的判别式为：

$$D = B^2 - 4AC$$

当  $D = \begin{cases} < 0 & \text{两个共轭复根} \\ = 0 & \text{两个相等实根} \\ > 0 & \text{两个不相等实根} \end{cases}$

程序可为：

```

PROGRAM ROOT
    IMPLICIT NONE
    INTEGER :: A,B,C
    REAL :: X1,X2,D,R,P
    PRINT *, 'INPUT A,B,C'
    READ *, A,B,C
    D=B*B-4*A*C
    first: IF D<0 THEN
        R=(-B)/(2*A)
        P=SQRT(-D)/(2*A)
        PRINT *, 'X1=',R,'+',P,'I'
        PRINT *, 'X2=',R,'-',P,'I'
    ELSE
        PRINT *, 'Equation have root of real'
    second: IF D=0 THEN
        R=(-B)/(2*A)
        PRINT *, 'Equation have equal of real'
        PRINT *, 'X1=X2=',R
    ELSE
        X1=(-B+SQRT(D))/(2*A)
        X2=(-B-SQRT(D))/(2*A)
        PRINT *, 'X1=',X1
        PRINT *, 'X2=',X2
    END IF second
    END IF second
    END IF first
END PROGRAM ROOT

```

例 4.4 求  $\arctgx + \arctgy$

解：

设  $x, y$  为弧度值， $sum$  为加和

则算法设计：

$$\arctg x + \arctg y = \begin{cases} \arctg \frac{x+y}{1-x \cdot y}, & \text{当 } x \cdot y < 1 \\ \pi + \arctg \frac{x+y}{1-x \cdot y}, & \text{当 } x > 0, x \cdot y > 1 \\ -\pi + \arctg \frac{x+y}{1-x \cdot y}, & \text{当 } x < 0, x \cdot y > 1 \\ \frac{\pi}{2}, & \text{当 } x \cdot y = 1 \end{cases}$$

程序可为：

```

PROGRAM SUM
IMPLICIT NONE
PARAMETER(pi=3.14159267)
REAL :: x,y,xy,sum
READ *,x,y
xy=x*y
first:IF abs(1-abs(xy))<1.0E-5 THEN
    sum=pi/2
ELSE
    sum=arctan((x+y)/(1-xy))
second:IF xy>1 THEN
third:IF x>0 THEN
    sum=sum+pi
ELSE
    sum=sum-pi
END IF third
END IF second
PRINT *, 'The sum of arctg x and arctg y is',sum
END PROGRAM sum.

```

## § 4.2 CASE 结构

对于多重选择问题，虽可用条件嵌套结构编程解决，但编制出来的程序往往是复杂的，又不太直观。特别是当多分支选择的各个条件由同一个表达式的不同结果值决定时，用 CASE 语句的结构来实现最有效。它的选择过程，很像一个多路开关，即根据表达式的不同值，分别执行不同的程序段。因此在程序设计中，它是一种强有力的手段。多路分支控制时，用 CASE 结构比 IF 结构编程具有更简明的特点。

### § 4.2.1 CASE 结构的一般形式

CASE 结构的一般形式为：

SELECT CASE(情况表达式)

```
CASE(情况选择 1)
```

```
语句块 1
```

```
CASE(情况选择 2)
```

```
语句块 2
```

```
...
```

```
[CASE DEFAULT
```

```
语句块 n+1]
```

```
END SELECT
```

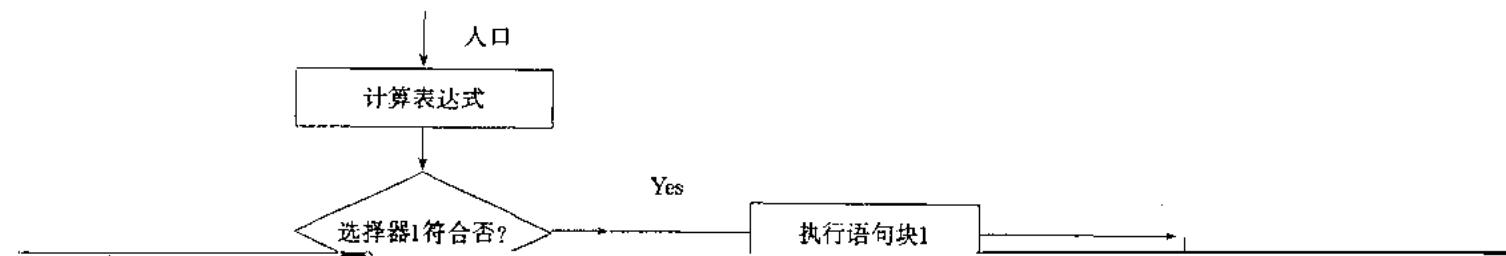
**功 能：**从多种情况中选择一种符合条件的情况去执行相应的语句块。

**说明：**①情况表达式只能是一个整型表达式或逻辑表达式或字符表达式，并且与后面的情况选择器之中的情况选择器的类型要一致。

②当所有 CASE(情况选择器 i)都不符合条件时,若含有可选择项 CASE DEFAULT 时,则执行 CASE DEFAULT 后面的语句块。CASE 结构中可以嵌套使用,也可以与 IF 结构相互嵌套,但必须层次清楚,不可交叉使用。

#### § 4.2.2 CASE 结构的控制执行

CASE 结构的控制可用图 4-3 表示。



控制执行从入口进入, 经过下列步骤:

- (1) 首先计算表达式的值
- (2) 当情况表达式的值与某一选择器 i 中的值相同时, 则执行该选择器 i 中的语句块 i, 然后转出口。
- (3) 当所有选择器中的值不符合情况表达式的值时, 若含有可选项 CASE DEFAULT 时, 则执行该语句的后续语句块(即 DEFAULT 块)。

#### 例 4.5 计算每月的天数。

解: (1) 算法分析: 根据天文历法规定, 凡是 1, 3, 5, 7, 8, 10, 12 各月天数为 31 天; 4, 6, 9, 11 各月中每月天数为 30 天; 而 2 月则按是否为闰年, 若闰年为 29 天, 否则是 28 天。

(2) 变量说明: yy——年, mm——月, day——天

(3) 程序:

```

PROGRAM Lenday
    IMPLICIT NONE
    INTEGER :: yy, mm, day
    PRINT *, INPUT yy, mm, '
    READ *, yy, mm
    SELECT CASE (mm)
        CASE(1,3,5,7,8,10,12)
            day=31
        CASE(4,6,9,11)
            day=30
        CASE(2)
            IF (yy mod 100. EQ. 0). AND. (yy MOD 400. EQ. 0)
                . OR. (yy MOD 100. NE. 0). AND. (yy MOD 4. EQ. 0)
            THEN
                day=29
            ELSE
                day=28
            END IF
        END SELECT
        PRINT *, 'mm=' , mm, 'day=' , day
    END PROGAM Lenday

```

#### § 4.2.3 CASE 结构的标识符

CASE 结构也分无名与有名两种形式, 上面所述为无名结构。有名的 CASE 结构, 其取名规则与一般变量名同, 标识结构名的方法也与标识 IF 结构名方法相同, 这里不赘述, 其有名结构的一般形式为:

结构名: SELECT CASE(情况表达式)  
CASE(情况选择 1)[结构名]

```

    语句块 1
CASE(情况选择 2)[结构名]
    语句块 2
    ...
CASE DEFAULT[结构名]
    语句块 n+1
END SELECT 结构名

```

其中：入口语句、出口语句的结构名要一致，CASE 语句的结构名可以省略。

例如：将学校中的学生、教师、干部规定不同的职责，输入计算机便于管理。我们可以把学生、教师、干部应尽的职责，分别编程为三个语句块，结构名为职责(DUTY)，情况表达式用职业(OCCUPATION)，而情况选择器的值，分别用学生、教师、干部表示。CASE 结构程序段可为：

```

DUTY:SELECT CASE (OCCUPATION)
    CASE ('STUDENT') DUTY
        语句块 1
    CASE ('TEACHER') DUTY
        语句块 2
    CASE ('CADRE') DUTY
        语句块 3
END SELECT DUTY

```

当情况表达式 OCCUPATION 取值为 'STUDENT' 时，执行语句块 1 (即学生职责范围详细内容)；当取值为 'TEACHER' 时，执行语句块 2，如此类推。

### § 4.3 循环结构

循环结构用来实现程序中某些语句序列按一定的规则多次重复执行。许多问题中需要用到循环结构的控制，例如：若干数据的求和，方程迭代求解，积分的计算，矩阵的计算等等，几乎所有的实用程序都包含循环。循环结构是三种基本结构(顺序、选择、重复)之一，它和顺序结构、选择结构共同作为各种复杂程序的基本构造单元。循环结构包含两个方面：

- 1) 重复对象：重复对象是一串语句，也可以是一个语句。重复的语句称为循环体。
- 2) 重复的控制条件，它用以控制重复执行。控制重复执行的条件可分为：
  - ① 当满足某个条件时重复；
  - ② 当满足某个条件后不再重复；
  - ③ 按给定的次数重复。

与此对应的 CM Fortran 提供了多种循环控制语句，下面分别加以介绍。

#### § 4.3.1 DO 循环结构

DO 循环结构的一般形式

[DO 结构名]DO[标号]循环变量 = 循环初值, 循环终值, 循环增值

### 循环体

[标号]END DO [DO 结构名]

其中:DO 是关键字,也是 DO 语句结构的入口;循环初值、终值、增值可以是整型或实型表达式,最好取整型,因实型可以带入误差,其值可正可负;可选项[DO 结构名]使用规则与条件结构名相当;可选项[标号]为正整数,是循环终止语句标号。

例如: DO 语句下列形式均是合法的:

DO 50 I=m,N \* L,J \* 3

DO I=-5,5,2

DO I=5.3,-0.2,-0.4

当增值是 1 时,可以省略。

DO 循环结构执行控制的过程可分为三步:

第一步:计算出循环初值、终值、增值,并对循环变量置初值;

第二步:判断循环变量值是否超过循环终值?若没有超过终值,往下执行循环体,否则退出循环。

第三步:执行循环一次后将增值加到循环变量中,并转移到第二步执行。

总的执行过程可由图 4-4 来描述。

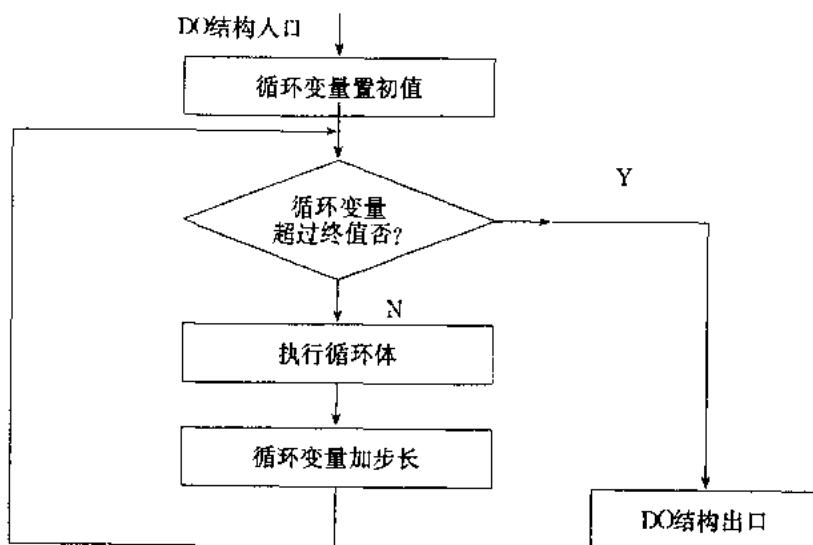


图 4-4

**例 4.6** 用辛普生法求:  $\int_0^1 (1+e^x) dx$  值。

解: ①算法

辛普生公式为

$$S \approx \frac{h}{3} [f(a) + f(b) + 4[f(a+h) + f(a+3h) + \dots + f(a+2n-1)h] + 2[f(a+2h) + f(a+4h) + \dots + f(a+(2n-2)h)]]$$

②变量说明

$a, b$  分别为积分上下限,  $h$  为区间长度,  $n$  为积分面积分成几等份。

### ③程序

```

PROGRAM CALCULATE — integral
IMPLICIT NONE
INTEGER::N,I
REAL::A,B,h,x,S2,S4,S
READ * ,A,B,N
H=(B-A)/(2*N)
X=A+H
S2=0.0
S4=1.0+EXP(X)
DO I=1,N-1,1
  X=X+H
  S2=S2+(1.0+EXP(X))
  X=X+H
  S4=S4+(1.0+EXP(X))
END DO
S=H*(1.0+EXP(A)+1.0+EXP(B)+4.0*S4+2.0*S2)/3.0
PRINT * 'S=' ,S
END PROGAM CALCUIATE—integral

```

#### § 4.3.2 DO TIMES 循环结构

DO TIMES 循环结构的一般形式:

```

DO [标号](整型表达式)TIMES
  循环体
  [标号]END DO

```

其中,整型表达式为重复循环的次数。

DO TIMES 循环结构语句的执行过程与前面 DO 循环结构语句功能和执行过程大致相同,所不相同的是 DO TIMES 结构中隐含了一个统计次数的变量,当执行循环体的次数达到整型表达式的值时,退出循环。

```

例如:      DO (N/10+1)TIMES
            语句块
            END DO
            DO 100 (N/100+1)TIMES
            语句块
            100 END DO
            OUT:DO(M * N)TIMES
            语句块
            END DO OUT

```

均是合法的。

### § 4.3.3 DO WHILE 循环结构

DO WHILE 循环结构的一般形式为：

DO [标号]WHILE(逻辑表达式)

    循环体

[标号]END DO

该语句功能是：首先判定逻辑表达式的值，若逻辑表达式值为真，则执行循环体；若逻辑表达式的值为假，则表示循环结束，跳出循环，执行循环语句以后的语句。图 4-5 说明了该语句的执行过程。

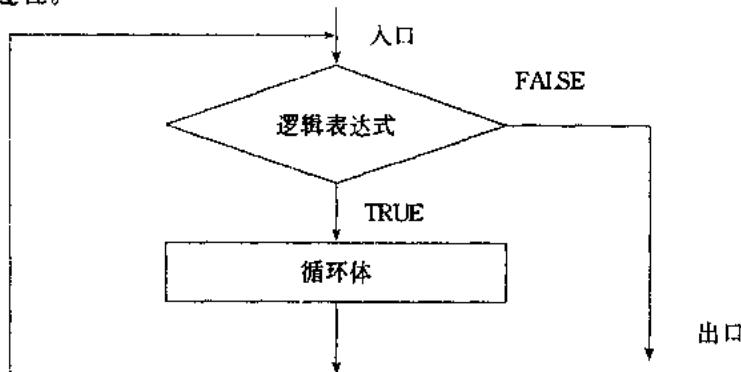


图4-5

注意：若第一次求得的逻辑表达式的值为 FALSE，则一次也不执行循环体，而直接执行该语句的后续语句。

**例 4.7** 求由键盘输入的 20 个数据的平均值。

解：程序可为： PROGAM AVERAGE

```

IMPLICIT NONE
INTEGER :: N
REAL :: SUM, X, AVE
SUM = 0.0
N = 1
DO WHILE (N .LE. 20)
    READ *, X
    SUM = SUM + X
    N = N + 1
END DO
AVE = SUM / (N - 1)
PRINT *, AVE
END PROGAM AVERAGE
  
```

**例 4.8** 求自然对数底( $e$ )的近似值。

解：①算法

近似值的计算公式如下：

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-1)!} + r_n$$

当余项  $r_n < \epsilon$  时停止计算, 编程时, 采用前后项之间的递推关系:  $x_n = x_{n-1} * \frac{1}{n}$

## ②程序

```

PROGRAM NATURAL
    IMPLICIT NONE
    INTEGER :: N
    REAL :: R, E
    E = 1; R = 1; N = 1
    DO WHILE R >= 1.0E-10
        E = E + R; N = N + 1; R = R / N
    END DO
    PRINT *, 'NATURAL LOGARITHM E = ', E
END PROGRAM NATURAL

```

从上述例子可以看出: 在进入 DO WHILE 语句前, 首先要考虑循环控制变量赋给正确的初值。循环体中, 必须放置修改循环变量值的语句, 修改循环变量值的语句要随着循环次数增加而逐渐趋近循环终止条件, 否则会造成死循环。

DO WHILE 语句适合于循环结束条件比较复杂, 并且循环次数未定的循环结构。

### § 4.3.4 EXIT 语句与 CYCLE 语句

EXIT 语句和 CYCLE 语句, 只能用在循环结构的循环体内, 在其它语句块内不能使用该语句。

#### 1. EXIT 语句

EXIT 语句的一般形式为:

EXIT[DO 结构名]

其中, 可选项 DO 结构名是指该循环结构中, 带有结构名的情况, 用来标明属于哪一层循环, 在多重循环结构标明结构名, 层次更加清楚。

EXIT 语句的功能: 停止循环的执行, 强迫退出本层循环结构体。

注意: 单独的 EXIT 语句没有实用意义, 它将使循环无条件中止。通常的情况下, EXIT 语句在循环体中与 IF 语句联系一起使用, 使满足一定条件, 退出循环。

例如:

```

DO WHILE(逻辑表达式)
    <语句序列 1>
    IF (E) THEN
        <语句序列 2>
        EXIT
    END IF
    <语句序列 3>
END DO

```

当 E 不满足时, EXIT 不起作用, 保持循环; 当 E 满足时执行 EXIT 语句, 停止循环。

## 2. CYCLE 语句

CYCLE[DO 结构名]

它的功能是在循环中跳过它下面的部分语句, 返回到循环第一个语句开始执行。CYCLE 语句的作用很类似于 END DO 语句, 但它只使本次循环的范围缩小。

例如:

```
DO WHILE(逻辑表达式)
    <语句序列 1>
    IF (E) THEN
        <语句序列 2>
    CYCLE
    ENDIF
    <语句序列 3>
END DO
```

当 IF…END IF 中的条件得到满足时, CYCLE 语句被执行, 此时不管 CYCLE 语句与 END DO 之间还有多少个语句尚未执行(即语句序列 3), 该次循环都将被强行中断, 控制转向循环入口语句的 DO WHILE。

**例 4.9** 用牛顿迭代法求  $x^5 - x^4 + 4x^2 - 1 = 0$  的近似根。

解: ①算法 牛顿迭代法的几何意义如图 4-6 所示。

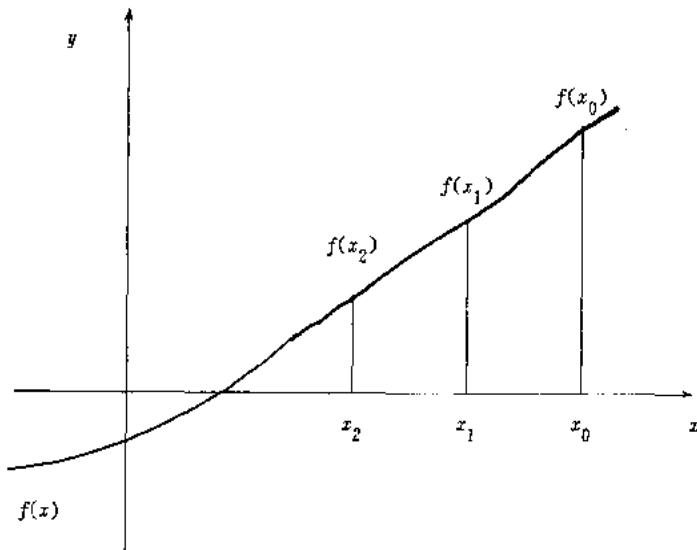


图 4-6

$$\text{切线的斜率为: } f'(x_n) = \frac{f(x_n)}{x_n - x}$$

$$\text{迭代公式为: } x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{或} \quad x = x_1 - f/F_1$$

这里  $x$  是新求得点在  $x$  轴上的坐标,  $x_1$  是上次求得的  $x$  轴上坐标,  $F = f(x) = x^5 - x^4 + 4x^2 - 1$ ,  $F_1 = f'(x) = 5x^4 - 4x^3 + 8x$ , 当两次迭代所得的近似根之差  $< 10^{-6}$  时, 视为满足求解的要求。

## ②程序

```

PROGRAM ROOT
    IMPLICIT NONE
    REAL :: X,X1,F,F1
    READ *,X
    DO WHILE .I.
        X1=X
        F=X**5-X**4+4*X**2-1
        F1=5*X**4-4*X**3+8*X
        IF(ABS(X-X1))>1E-6 THEN
            CYCLE
        ELSE
            PRINT *,'X=' ,X
            EXIT
        ENDIF
    END DO
END PROGRAM ROOT

```

## § 4.3.5 循环结构的嵌套

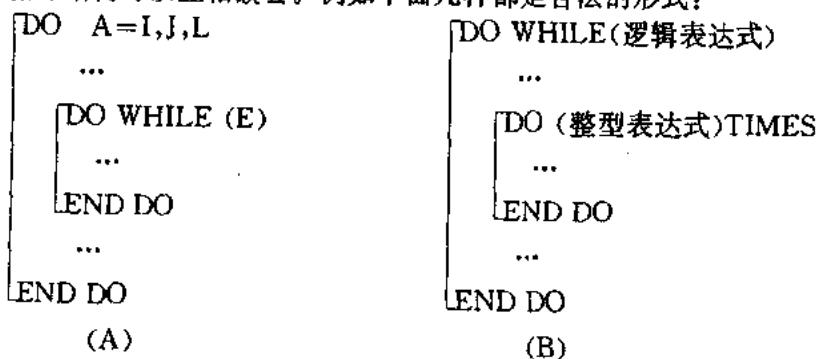
一个循环体内又包含另一个完整的循环结构，称为循环结构的嵌套。外层的循环称为外层循环，内层的循环称为内循环。若内循环体中还可以嵌套循环，这就是多层循环嵌套。循环嵌套的概念对各种语言都是一样的。例如下列是一个两重循环嵌套：

```

first:DO I=1,9,1
second:DO J=1,9,1
    L=I * J
    PRINT * ,I,' * ',J,' = ',L
END DO second
END DO first

```

各种循环结构可以互相嵌套。例如下面几种都是合法的形式：



```
DO(整型表达式)TIMES
  ...
  DO WHILE
    ...
    END DO
  ...
END DO
(C)
```

```
DO WHILE (e1)
  ...
  DO WHILE (e2)
    ...
    END DO
  ...
END DO
(D)
```

在书写结构比较复杂而嵌套层次较多的程序时,为了使阅读程序层次分明,结构清晰,最好是把程序写成锯齿状。如上面各式所示,把需要匹配的语句左端对齐,使匹配情况和嵌套关系一目了然,避免嵌套层次不匹配。在嵌套内若有 EXIT 语句和 CYCLE 语句时,最好带上 DO 结构名使程序便于理解。例如下列不同循环结构嵌套的程序段:

```
SUM=0
READ (IUN) N
-----
```

```

CALL CALCULATE (ARRAY(I),RESULT)
IF(RESULT.LT.0.0)CYCLE
    ! don't sum negatives
SUM=SUM+RESULT
IF (SUM.GT.SUM-MAX) GOTO 81
80 CONTINUE ! shared by both loops
81 CONTINUE

```

循环结构嵌套时规定：

- (1) 循环的嵌套不得产生交叉。
- (2) 多重循环中，各个层的循环变量不得同名。
- (3) 各层的循环体都应是完整的，且均不允许从循环体外直接转向循环体内。
- (4) 多重循环根据实际问题可共享一个终端语句。

例如：下面用法是正确的。

```

DO 20 I=1,9,1
    DO 20 J=1,9,I
        L=I * J
20      PRINT *,I,'*',J,'=',L

```

该程序段和下列语句功能完全一样。

```

DO 10 I=1,9,1
    DO 20 J=1,9,I
        L=I * J
        PRINT *,I,'*',J,'=',L
20      END DO
10      END DO

```

共享循环终端语句的一般形式为：

```

DO 100 WHILE ( $e_1$ )
    <语句序列 1>
DO 100 WHILE ( $e_2$ )
    <语句序列 2>
DO 100 WHILE ( $e_3$ )
    <语句序列 3>
100      <共享循环终端语句>

```

共享循环终端语句的引入，往往能使程序层次更加清楚，结构更加简单。

**例 4.10** 打印由 1,2,3,4, 这四个数字组成的所有可能的四位数，并统计它们的个数（允许各位数字相似数：如 1111,2222, …）。

解 程序段可为：

```

N=0
DO 10 I=1,4,1
    DO 10 J=1,4,1

```

```

DO 10 K=1,4,1
    DO 10 L=1,4,1
        M=I * 1000 + J * 100 + K * 10 + L
        PRINT *,M
10           N=N+1;PRINT *,'N=' ,N

```

**例 4.11** 利用枚举法求解方程  $x^3+y^3+z^3=8$  在  $-2 \leq x \leq 2, 0 \leq y \leq 3, -3 \leq z \leq 4$  区间上所有的整数根。

解：由于求解的是整数根，可用枚举法将  $x, y, z$  在其区间内的所有整数代入方程进行验证，求得方程的整数根。

程序可为：

```

PROGAM ROOT
    IMPLICIT NONE
    INTEGER :: x,y,z,M
    DO 10 x=-2,2,1
        DO 10 y=0,3,1
            DO 10 z=-3,4,1
                M=x ** 3+y ** 3+z ** 3
                IF(M.EQ.8)THEN
                    PRINT *,x,y,z
                END IF
10    CONTINUE
END PROGAM ROOT

```

#### § 4.3.6 隐含 DO 循环

隐含 DO 循环实际上是带控制循环变量的 DO 结构。但简化成只有 DO 结构的第一句并且把 DO 关键字隐去。隐含 DO 循环的一般形式：

(I/O 表列, 循环变量 = 循环初值, 循环终值, 循环增值)

功能：只能作为读写语句的输入输出表中的一个组成部分，用来控制重复读写的次数。它不是单独语句。

其中：I/O 表列是输入输出表，而 I/O 表列还可以包含隐含 DO 表。输入输出表中或是输入表项，或是输出表项。输入表项必须是下列之一：变量名、数组元素名、字符串名和数组名。而输出表项必须是下列之一：变量名、数组元素名、字符串名、数组名和表达式。

隐含 DO 循环的执行是按照与 DO 循环结构的规则给“I/O 表”中的内容进行输入输出处理。

例如：READ \*,(A(I),I=1,3)

相当于给 A(1),A(2),A(3) 通过键盘输入数据，也相当于：

```

DO I=1,3,1
    READ *,A(I)
END DO

```

当“I/O 表列”中有变量名和数值元素同时出现时，应将它们视为每次循环的一个循环整体。

例如：READ \* , (X,A(I),I=1,3)

这时输入应按 X,A(1),X,A(2),X,A(3) 的顺序进行。

使用隐含 DO 循环有下列各种形式：(以输出为例)

(1) 按指定步长读写，如：

PRINT \*, (A(I),I=2,30,2)

则打印 A(2),A(4),…,A(30) 的值

(2) 隐含表与普通表混合，如：

PRINT \*, A,B,(C,I=1,5),D

则打印完 A,B,5 个 C 的值，再打印 D 的值。

(3) 两个隐含 DO 表，如：

PRINT \*, (A,I=1,5),(B,J=1,6)

则打印完 5 个 A 值，再打印 6 个 B 值。

(4) 隐含 DO 循环的嵌套，要遵循多重循环对循环嵌套的全部要求。

例如：PRINT \*, ((A(I,J),J=1,3),I=1,2)

则打印 A(1,1),A(1,2),A(1,3),A(2,1),A(2,2),A(2,3) 的值。

它相当于：

DO 10 I=1,2,1

DO 10 J=1,3,1

10 PRINT \*, A(I,J)

注意，在利用隐含 DO 循环作为输入时，循环变量不能出现在“I/O 表列”中。例如，下面用法是不允许的：

READ \*, (I,A(I),I=1,5)

而且若初值、终值、增值用变量表示时，则该变量也不允许出现在“I/O 表列”中。例如，下面用法是不允许的：

READ \*, (I,A(N),N=I,10)

## 第五章 CM Fortran 数组与数据处理

迄今为止,我们在程序中使用的变量都是各自独立的,相互之间没有什么内在的联系。例如:A,B,C,X3等。我们把这种变量称为离散型变量。在科学计算和数据处理中,经常会遇到按一定顺序排列的量,并且这些量具有相同的数据类型。这时如果仍用离散型变量表示,则必须用许多变量才能表示它们,这是很不方便的。为了便于处理大批量的同类型数据,我们有必要将众多变量组成一定的结构。

例如:多项式  $a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$  的系数项可表示为一向量  $(a_0, a_1, \dots, a_n)$  该向量本身就是按  $a_i$  在多项式中的次序而排列的。通常,我们把一组按一定顺序排列的具有相同类型结构的数据称为数组。上述多项式系数若用数组 A(I) 表示,  $(a_0, a_1, \dots, a_n)$  分别是数组 A(I) 中的每个元素,此处的 A 称为数组名,而  $a_i$  是数组 A(I) 中的一个元素。数组的引入简化了程序的书写,增强了程序的可读性,它是程序设计中最重要的结构之一。本章详细讨论与数组相关联的语句应用和数据处理技术。

### § 5.1 数组的定义和有关说明

#### § 5.1.1 数组的定义和数组说明符

数组定义:数组是一批相同类型数据的线性组合。

数组代表一批数据,并且要求这些数据必须具有相同的类型。例如都是整型、实型、逻辑型、字符型等等。数组也有类型,它的类型就是数组内所存放的数据的类型。数组是一批类型相同的数据的线性组合,即数组内的每个数据都有确定的先后顺序。并且每个数据都有对应的编号,以示区别数据之间的前后关系。数组有数组名,数组名同变量名的命名规则相同。当要表示数组中某一具体的数据时,则需通过数组中的数组元素来指定,即数组中的每个元素称为数组元素(也称下标变量)。数组元素的表示方法,是在数组名后加一括号,括号内注明下标值。

数组元素的作用同普通变量的作用一样,所不同的是数组元素是隶属于所在的那个数组,它的性质也是由所在的数组来决定的。如果数组被定义为整型数组,则该数组中所包含的所有数组元素也为整型的。要描述一个数组,必须指出数组的名字、数组元素的类型、数组的维数及每维所对应的维界等,这些内容由数组说明符来完成。

数组说明符:在一个程序单位内,数组说明符规定了数组名,同时也规定了该数组的某些特性。在一个程序单位内,一个数组名只允许有一个数组说明符。

数组说明符的一般形式为: A(d[,d]…)

其中:A 是数组名;d 是维说明符,又称下标说明符,下标与下标之间用逗号分隔。一个数组

中下标的多少，则表示该数组中维数的多少。

当数组说明符中，只含一个下标时，我们说该数组是一维数组；当数组说明符中含有两个下标时，则称该数组是二维数组；当数组说明符中包含三个下标时，则称该数组是三维数组，…。例如：若在数组说明部分，说明了数组 A(99), B(8,9), C(3,6,9), D(2,4,8,10)，则 A 表示是一维数组，B 表示是二维数组，C 表示三维数组，D 表示四维数组。

数组的维数由数组说明符中的维说明符来确定。数组最小的维数是 1，最大的维数不得超过 7。

维说明符的一般形式是： $[d_1 : ]d_2$

其中： $d_1$  是维下界， $d_2$  是维上界。

每一个维说明符都对应一个维界，若仅指明上界  $d_2$ ，则其下界  $d_1$  的缺省值为 1。维上界和维下界都是算术表达式，在该表达式中所有的常数、常数符号名和变量都应是整型的。对于任何维界来说，其中的数值可以为正、负或零。然而，维上界必须大于或等于维下界值。维的大小是指数组中维内有多少排序号，通常称为维长。维长的值等于： $d_2 - d_1 + 1$ 。

若数组说明符定义了 A, B 两个数组分别是：A(1:500), B(-49:450) 时，则数组 A 中的维长为： $500 - 1 + 1 = 500$ ；而数组 B 中的维长度为： $450 - (-49) + 1 = 500$ 。

由上述可知，在一维数组中维长就是该数组元素的个数。

### § 5.1.2 数组说明语句

在 CM Fortran 程序中，如果使用数组，则必须用数组说明语句对所使用的数组加以说明，否则在编译时计算机就会发出错误信息。

在 CM Fortran 中，数组说明语句有三种形式：

#### 1. 用 DIMENSION 语句来进行数组说明

DIMENSION 语句可以说明一个或多个数组的维数、形状、大小等。DIMENSION 语句像所有说明语句一样，必须放在本程序块的所有可执行语句的前面。

DIMENSION 语句的一般形式为：

DIMENSION 数组名(数组形状说明)[, 数组名(数组形状说明)]…

其中：①数组名的规定与变量名的规定相同，数组名服从 Fortran 中隐含的类型规则（即 I-N 规则）。因此一个数组名以 I, J, K, L, M, N 字母开头时，它的数组元素只能存放整型数值；而除此六个字段以外，以其它字段开头的数组名，则数组中只能存放实型数组（注：若要使数组名不服从该隐含规则时，就用其它方式来进行数组说明，并同时指出数组元素的类型）。为了使 CM Fortran 能像其它高级语言一样，取消隐含规则，以免隐含规则造成人们容易忽视提供的数据类型与隐含规则不一致的错误。因此，CM Fortran 提供了取消隐含规则语句，即 IMPLICIT NONE 语句。它向系统声明不许用隐含说明，该语句写在主程序名后或子程序名的后继行上。②数组形状说明是指定了该数组的维数，每维的长度，每维的上下界以及大小等（注：关于数组的不同形状说明，在后面将专门介绍）。

DIMENSION 语句举例说明如下：

DIMENSION A(10), B(10,70)

DIMENSION C(-3:12, \*)

DIMENSION D(:, :, :, :, :, :)

第一个数组说明语句,说明数组 A 和 B 是显式形状的实型数组,其中 A 为一维数组,它的下界值是 1,上界值是 10,A 数组的大小等于 10。而 B 为二维数组,其维界分别为 1:10,1:70,数组的大小为  $10 * 70 = 700$ ,而 B 数组的形状是[10,70]。

第二个数组说明语句,说明数组 C 是假定大小的实型数组,规定了 C 是二维数组。数组 C 中第二个下标是未知的,所以该数组的形状和大小是不确定的。

第三个数组说明语句,说明数组 D 是假定形状的实型数组,规定了数组 D 是五维数组,但该数组中的维界、形状、大小等都是未定的。

### 2. 用类型说明语句来进行数组说明

在数组说明中也可以直接用类型说明语句来说明数组的维数、形状和大小等。此时的数组名也和变量名一样,不再遵循隐含规则。

用类型说明语句来说明数组的一般形式为:

类型说明 数组名(数组形状说明)[,数组名(数组形状说明)]…

其中,类型说明可以是整型,实型,逻辑型,字符型等;数组名和数组形状说明与上述相同。

类型说明语句对数组说明的例子如下:

```
INTEGER A(100,-2:2),B(2,2,3,2,2)
REAL C(2:6,7:8),D(3:8,1:5)
LOGICAL E(3,2),F(2:7,3:8)
```

第一个数组说明语句,说明数组 A 和 B 为整型数组,并且规定了 A 为二维整型数组,A 数组的形状是[100,5],A 数组的大小为 $(100-1+1) * (2-(-2)+1) = 500$ ;而规定了 B 数组为 5 维数组,B 数组的形状是[2,2,3,2,2],它的大小是: $2 * 2 * 3 * 2 * 2 = 48$ 。

第二个数组说明语句,说明数组 C 和 D 为二维的实型数组,并且规定了数组 C 的大小是: $(4-2+1) * (8-7+1) = 10$ ,C 数组的形状是[5,2]。而数组 D 的大小是: $(8-3+1) * (5-1+1) = 30$ ,数组 D 的形状是[6,5]。

第三个数组说明语句,说明数组 E 和 F 为二维的逻辑类型数组,数组元素中所存放的数据都为逻辑值。数组 E 的大小是: $(3-1+1) * (2-1+1) = 6$ ,数组 E 的形状是[3,2]。数组 F 的大小是: $(7-2+1) * (8-3+1) = 36$ ,数组 F 的形状是[6,6]。

### 3. 利用数组属性来进行数组说明

数组属性说明规定该数组应具备的所属性质,这些性质包括:数组的类型,维数,形状和大小等。

数组属性说明语句的一般形式为:

类型说明,数组属性说明表::数组名[(数组形状说明)]…

其中:数组属性说明表包括:①必选项——数组属性说明符 ARRAY(数组形状说明);②可选项——SAVE 和 DATA,可选项 SAVE 的功能是当程序段返回或结束时,计算机中仍保留该程序段中数组元素值。可选项 DATA 的功能是规定了所说明的数组同时获得数组元素的初值。

数组属性说明语句的简单例子为:

```
REAL,ARRAY(-5:5)::x,y,z
```

该语句说明了三个显式形状的实数组 x,y,z 它们都是一维数组,数组形状和大小均相同。它们的维界均有下界是 -5,上界是 5,数组的维长均是 11。

在数组属性说明语句中,某些数组名也可以跨越数组属性的说明,即不属该数组属性说明的限制。例如:

```
INTEGER, ARRAY(10)::A,B,C(5,5)
```

该语句说明了三个显式形状的整型数组,数组 A 和 B 均是一维数组,C 是二维数组。即数组 C 跨越了数组属性说明是一维数组的限制,另外定义了数组 C 的维数、形状和大小。

数组属性说明语句的其它形式举例如下:

```
REAL,ARRAY(10),SAVE,DAVE::A=[1:10]
```

! A 为显式形状的一维实型数组,而且数组元素的初值从 1 到 10。

```
INTEGER,ARRAY(10,10),SAVE::A,B,C
```

! A,B,C 均为显式形状的二维整型数组。

```
REAL,ARRAY(N,10)::A,B
```

! 数组 A 和 B 为实型的二维可调数组。

```
REAL,ARRAY(:,)::x,y
```

: 数组 x 和 y 为实型的一维假定形状数组。

```
REAL,ARRAY(N,*)::s
```

! 数组 s 为实型的二维假定大小的数组。

```
LOGICAL,ARRAY(5,3)::MASK1,MASK2
```

! 数组 MASK1 和 MASK2 均为显式形状的二维逻辑型数组。

综上所述,数组说明的三种形式功能相同,但略有差异。在数组说明时,根据各人的习惯和需要选择其中任一种进行说明,不得重复说明,否则就是错误的。

### § 5.1.3 数组的下标与存储次序

CM 数组中的数组元素的下标可以是整型常数,整型变量,整型表达式,也可以是整型向量(注:关于下标用向量表示的情况,将在后面介绍)。下标规定了数组元素在数组中的排列次序,因此只要指出数组名和下标值,就可以确定某个数组元素。即,数组的一个重要性质就是可以通过下标来访问数组中的每一个数组元素。在访问一个数组元素时,首先是对下标中的下标表达式求值,然后根据下标表达式的值,计算机自动选取该数组中对应的数组元素。

数组在计算机内,占据一片连续的存储单元。数组元素的存储次序按数组的下标线性地顺序存放。因此,我们可以用下标值来描述数组元素的存放次序。如果有一个数组元素要引用,可根据下标次序值的计算公式,求出数组元素的下标。

下标次序值即标明了该数组元素在该数组中按存放次序是第几个。当下标次序值是 1 时,就表示它是数组中的第一个元素,依此类推。

在表 5.1 中,数组的维数 n 大于等于 1 且小于等于 7。 $j_i$  是第 i 维的下界, $k_i$  是第 i 维的上界。若只指明了维上界  $k_i$ ,则相应的维下界  $j_i$  是 1。 $s_i$  是指第 i 个下标表达式的整型值。 $d_i$  是指第 i 维的大小, $d_i = k_i - j_i + 1$ 。

在 CM Fortran 中规定数组在机内存储,按列的顺序来存放。即先存完第一列内诸元素,接着存放第二列内诸元素,然后存第三列,直到最后一列存完为止。

例如:有如下二维数组 A:

表 5.1 下标次序值的计算公式

维数 n	显式形状说明	下标表列	下标次序值	最大下标值 (数组体积)
1	(j <sub>1</sub> :k <sub>1</sub> )	(s <sub>1</sub> )	1+(s <sub>1</sub> -j <sub>1</sub> )	d <sub>1</sub>
2	(j <sub>1</sub> :k <sub>1</sub> , j <sub>2</sub> :k <sub>2</sub> )	(s <sub>1</sub> , s <sub>2</sub> )	1+(s <sub>1</sub> -j <sub>1</sub> ) + (s <sub>2</sub> -j <sub>2</sub> ) * d <sub>1</sub>	d <sub>1</sub> * d <sub>2</sub>
3	(j <sub>1</sub> :k <sub>1</sub> , j <sub>2</sub> :k <sub>2</sub> , j <sub>3</sub> :k <sub>3</sub> )	(s <sub>1</sub> , s <sub>2</sub> , s <sub>3</sub> )	1+(s <sub>1</sub> -j <sub>1</sub> ) + (s <sub>2</sub> -j <sub>2</sub> ) * d <sub>1</sub> + (s <sub>3</sub> -j <sub>3</sub> ) * d <sub>2</sub> * d <sub>1</sub>	d <sub>1</sub> * d <sub>2</sub> * d <sub>3</sub>
...	...	...	...	...
7	(j <sub>1</sub> :k <sub>1</sub> , j <sub>2</sub> :k <sub>2</sub> , ..., j <sub>7</sub> :k <sub>7</sub> )	(s <sub>1</sub> , s <sub>2</sub> , ..., s <sub>7</sub> )	1+(s <sub>1</sub> -j <sub>1</sub> ) + (s <sub>2</sub> -j <sub>2</sub> ) * d <sub>1</sub> + (s <sub>3</sub> -j <sub>3</sub> ) * d <sub>2</sub> * d <sub>1</sub> ... + (s <sub>7</sub> -j <sub>7</sub> ) * d <sub>6</sub> * ... * d <sub>1</sub>	d <sub>1</sub> * d <sub>2</sub> * ... * d <sub>7</sub>

$$A = \begin{bmatrix} A(1,1) & A(1,2) & \cdots & A(1,n) \\ A(2,1) & A(2,2) & \cdots & A(2,n) \\ \cdots & & & \\ A(n,1) & A(n,2) & \cdots & A(n,n) \end{bmatrix}$$

数组元素的存放顺序是:A(1,1), A(2,1), ..., A(n,1), A(1,2), A(2,2), ..., A(n,2), ..., A(1,n), A(2,n), A(3,n), ..., A(n,n)。

因此,在输入数组的数据时,按输入的先后次序,先输入第一列的(竖向)各元素,再输入第二列,第三列...。数组按列存放的方式与数学上按行处理习惯不同,这是Fortran的一个特性,要特别注意。只有按列顺序输入数组的各元素,计算机内收到的数组才是正确的。

**例 5.1** 假设 A 和 B 是一维的实型数组,数组说明为:REAL A(9),B(-4:5),则它们各自按数组下标值的大小,由小到大顺序排列存储在一片连续的存储空间内。可表示如下:

数组 A	数组元素	A(1)	A(2)	A(3)	...	A(9)
	下标次序值	1	2	3	...	9

数组 B	数组元素	B(-4)	B(-3)	B(-2)	...	B(5)
	下标次序值	1	2	3	...	10

**例 5.2** 假设 C 为二维整型数组,数组说明为:INTEGER C(-2:2,0:1),则数组 C 根据下标次序值按列顺序地存储在一片连续的存储空间内。可表示如下:

数组 C	数组元素	C(-2,0)	C(-1,0)	C(0,0)	C(1,0)	...	C(2,1)
	下标次序值	1	2	3	4	...	10

从上述的例 5.2 和数组的下标次序值的计算公式可以看出:若有一个数组 A 说明为:  
 $A(j_1, k_1, j_2, \dots, k_2, \dots, j_r, k_r)$ , 则数组 A 的第一个下标次序值是 1 的数组元素一定是  $A(j_1, j_2, \dots, j_r)$ 。而它的直接后继应为  $A(j_1+1, j_2, j_3, \dots, j_r)$ , 并且下标次序值是 2。也就是说按列存放是先将数组的第一维按递增次序增加, 直到它到达该维的上界  $k_1$  时, 然后将直接的后继第二维加 1, 第一维又重新从  $j_1$  开始递增。即  $A(k_1, j_2, j_3, \dots, j_r)$  的下一个元素为  $A(j_1, j_2+1, j_3, \dots, j_r)$ 。当第二维也到达了它的维上界  $k_2$  时, 则将第三维加 1, 然后第一维第二维又从  $j_1, j_2$  开始递增。多维数组的排列顺序依此类推, 那么它的最后一个数组元素一定是  $A(k_1, k_2, k_3, \dots, k_r)$ 。

#### § 5.1.4 数组段(部分数组)

在 CM Fortran 中, 不仅可以直接对数组名进行赋值和直接将数组名参加各种运算操作, 而且还可以对数组中的某一部分进行赋值或参加各种运算操作。我们把数组中的部分元素称为数组段(或称部分数组)。即数组段中的元素是该数组中全部元素的一个子集。或简称数组段是数组的一个子集。数组段是按照数组每一维的下标规定, 从原数组中选取部分元素的结果。凡是整个数组能用的地方, 都能使用数组段。例如: 若数组 A 是一个  $10 \times 10$  的矩阵, 则称  $A(1:5, 1:5)$  是数组 A 的一个数组段。数组段  $A(1:5, 1:5)$  是数组 A(10,10) 的左上象限, 而数组段  $A(1:5, 6:10)$  是数组 A 的右上象限, 而且这些数组段都是  $5 \times 5$  的矩阵。因此, 语句  $A(1:5, 1:5) = A(1:5, 6:10)$  是把 A 中的右上象限元素拷贝到左上象限中, 数组 A 中其它的元素保持不变。这种数组段中的元素特征是, 选取原数组中一段相连的元素组成数组段中的元素。而数组段  $A(1:10:3, 1:10:3)$  是一个  $4 \times 4$  的矩阵, 数组段中的元素是:

$$\begin{array}{cccc} A(1,1) & A(1,4) & A(1,7) & A(1,10) \\ A(4,1) & A(4,4) & A(4,7) & A(4,10) \\ A(7,1) & A(7,4) & A(7,7) & A(7,10) \\ A(10,1) & A(10,4) & A(10,7) & A(10,10) \end{array}$$

在数组段  $A(1:10:3, 1:10:3)$  中, 行下标和列下标均用三个数据表示。我们把一个下标中使用三个数据来表示下标变化情况, 称为三元组下标。在一个三元组下标中的第一个数据表示该数组段中某维所选元素的起始下标值, 第二个数据表示该维所选元素的终止下标值, 第三个数据表示该维所增加的步长值。这种三元组下标所组成的数组段中的元素的特征是: 按照所给的增量的步长值, 有规律地间隔选取原数组中的部分元素组成的集合。

在数组段中, 还提供了一种特别有用向量值下标。向量值下标的运用, 使得并行计算机如虎添翼, 不仅缩短了并行数据程序的运行时间, 而且提高了并行计算机的使用效率。如果在一个下标向量里多次出现同一个索引, 那么同一个源值就会和多个目的地进行通信。

例如: 若执行下列语句

$V = [1, 2, 2, 3, 3, 3, 1, 2, 3, 4, 5, 6, 6, 6, 1, 1, 2, 2, 3]$

S=[10,20,30,40,50,60]

A=S(V)

则 A 的值将是：

[10,20,20,30,30,30,10,20,30,40,60,60,60,10,10,20,20,30]

又如：假定 P 是一个长度为 N，由整数 1 到 N 组成的向量。那么 P 是一个置换向量，语句 B=B(P)是对数组 B 进行一个置换。

下面分别介绍数组段中的三元组下标和向量值下标。

### 1. 数组段中的三元组下标

若将一个数组中有规律地间隔选取部分元素来组成一个数组段，则这种有间隔功能的数组段的下标就是三元组下标。

一维数组中三元组下标的一般形式为：

数组名([表达式 1]:[表达式 2][:表达式 3])

其中：[表达式 1]:[表达式 2][:表达式 3]称为三元组下标。在三元组下标中，[表达式 1]是表示所选元素的起始下标值，[表达式 2]是表示所选元素的终止下标值，[表达式 3]是表示有规律选取数组中元素的跨度(也称增值步长)。整个三元组下标就相当于循环控制变量中的初值、终值、步长值。

例如：假设数组 A 定义为一维数组，且维长是 30。若任选一个数组段 A(3:30:3)，则三元组下标的起始下标值是 3，终止下标值是 30，下标的增值是 3。

数组段 A(30:30:3)中的元素，是由 A(3),A(6),A(9),A(12),A(15),A(18),A(21),A(24),A(27),A(30)所组成的集合。

若选取数组段为 A(2:20:3)，则三元组下标的起始下标值是 2，终止下标值是 20，下标增值是 3。数组段内的元素，是由 A(2),A(5),A(8),A(11),A(14),A(17),A(20)所组成的集合。

在三元组下标的一般形式中，方括号中的任一项均是可选项，即任一项均可省略不写。

当三元组下标中的第一个表达式为缺省时，则表示原数组中的维下界，默认为三元组下标的起始下标值。例如：若有数组 B 说明如下：

INTEGER B(5:50)

则程序执行部分中允许出现数组段 B(:,30:5)。在这里，三元组下标中第一个表达式缺省，此时三元组下标的起始下标值默认为是 5。因此，数组段 B(:,30:5)中的元素，是由 B(5),B(10),B(15),B(20),B(25),B(30)所组成的集合。

当三元组下标中的第二个表达式为缺省时，则表示原数组中维上界默认为该数组段的三元组下标的终止下标值。

当三元组下标中的第三个表达式为缺省时，则表示跨度默认为 1，从三元组的起始下标值到终止下标值都是连续的，这就成为开始介绍数组段的情况。

上面所述的数组段中的元素，都是从原数组中顺序选取组成的元素集合。数组段中元素的排列顺序，与数组元素排列顺序一致。即三元组下标中的第一个表达式值小于等于第二个表达式的值，也就是说第三个表达式的值为正值。正如循环控制变量中的步长值可取正值，也可以取负值。同样的道理，三元组下标中的第三个表达式值也可以为负值，此时三元组下标中第一个表达式值是大于第二个表达式的值。在这种情况下的数组段中的元素排列顺序，

与原数组中元素排列顺序相反。

例如：假设数组 C 说明为一维数组，且维长是 28。任选一个数组段 C(18:3:-3)，则此三元组下标的起始下标是 18，终止下标值是 3，下标的增值是 -3。数组段 C(18:3:-3) 中的元素排列顺序是：C(18), C(15), C(12), C(9), C(6), C(3) 共 6 个元素组成有序集合。

当数组是二维时，数组段中是由两个三元组下标表示的。即：

二维数组中三元下标的一般形式为：

数组名([表达式 1]:[表达式 2][:表达式 3],[表达式 1']:[表达式 2'][,表达式 3'])

其中：逗号前的三个表达式称为二维数组行下标的三元组，逗号之后的三个表达式称为二维数组列下标的三元组。

例如：假设数组 D 是一个  $10 \times 10$  的矩阵。若任选一个数组段 D(2:8;2,3:9;3)，则数组段中第一个三元组下标表示二维数组行下标变化范围；数组段中第二个三元组下标表示二维数组列下标变化范围。数组段此时选取元素规则是：先将列下标定在起始下标值上，然后遍历行下标，顺序选取各元素。这时再把列下标增加一步长值，又重新遍历行下标，如此重复，直到全部选完为止。这种选取元素的方法相当于把行下标看成是内循环变量变化，列看成是外循环变量变化。如对数组段 D(2:8;2,3:9;3) 中元素的选取顺序是：第一遍选取元素为：D(2,3), D(4,3), D(6,3), D(8,3)；第二遍选取元素为：D(2,6), D(4,6), D(6,6), D(8,6)；第三遍选取元素为：D(2,9), D(4,9), D(6,9), D(8,9)。即数组段 D(2:8;2,3:9;3) 中的元素组成三行四列的矩阵。

关于三维数组以上的数组段，我们仅以一个简单的例子说明之。如果一个数组 E 说明 E(5,4,3)，则有一个数组段是：E(3:5,2,1:2)。数组段 E 所选取的元素构成一个三行两列的形状，全部所选元素如下所示：

E(3,2,1)	E(3,2,2)
E(4,2,1)	E(4,2,2)
E(5,2,1)	E(5,2,2)

由此可见，在数组段中，某个三元组下标也可以被一个整型值代替，它表示在该维始终取该值作为下标，其它各维则仍变化如前。

## 2. 向量值下标

在数组段中，我们不仅可以使用三元组下标，从已知的数组中有规律顺序选取数组中的元素组成一个数组段。而且也可以使用向量值下标，从已知的数组中任意地选取数组中的元素组成一个数组段（向量值下标表示的数组段也称向量数组）。向量值下标的显著特点是：任意地和重复地选取已知数组中的元素。向量值下标可以代替数组段中的部分三元组下标，也可以在数组段中完全由向量值下标来表示。

### (1) 向量值下标作为数组段中的部分三元组下标

**例 5.3** 假设有  $10 \times 10$  的二维数组 AA，有向量  $V = [2, 4, 2]$ 。若给出下列数组段：

AA(2:8;3,V)

则数组段中所选元素的顺序是：第一遍选取的元素是 AA(2,2), AA(5,2), AA(8,2)；第二遍选取的元素是 AA(2,4), AA(5,4), AA(8,4)；第三遍选取的元素是 AA(2,2), AA(5,2), AA(8,2)。数组段中全部元素组成 3 行 3 列的阵列，形状如下所示：

AA(2,2)      AA(2,4)      AA(2,2)

AA(5,2)	AA(5,4)	AA(5,2)
AA(8,2)	AA(8,4)	AA(8,2)

从上例中可以看出,当数组中使用了向量值下标时,向量值下标内各数值的先后选取,仍然类似于三元组下标的变化情况。当向量值下标作为第二维下标时,首先确定向量 V 中第一个值 2,第一维中的行下标按照三元组下标变化规律选遍行下标中下标值;然后确定向量 V 中的第二个值 4,行下标又重新变化选遍行下标中下标值;最后确定向量 V 中第三个值 2,行下标再次变化选遍行下标中各下标值,直到最后全部选完为止。

反之,若向量值下标是作为二维数组段中的行下标时,则首先把列下标中三元组下标当作外循环变量变化看待,确定在起始值上,而后选遍行下标中由向量值下标列出的各值;再将列下标中的三元组下标增加一步长值,又重新选遍向量值下标内列出的各值;如此重复,直到最后全部选完为止。

值得注意的是:向量值下标内的各值,必须是明确规定过的值。而且向量值下标内的各值只有先后位置关系,不存在大小顺序。

#### (2) 向量值下标作为数组段中的全部下标

**例 5.4** 假定 Z 是形状[5,7]的二维数组,U 和 V 均是一维数组并且形状分别是[3]和[4]。

若给定的 U 和 V 是:U=[1,3,2],V=[2,1,1,3],则数组段 Z(U,V)的全部元素由下列各元素组成:

Z(1,2)	Z(1,1)	Z(1,1)	Z(1,3)
Z(3,2)	Z(3,1)	Z(3,1)	Z(3,3)
Z(2,2)	Z(2,1)	Z(2,1)	Z(2,3)

数组段 Z(U,V)中各元素的选取方法,如前所述,在这里不再赘述。

## § 5.2 数组的赋值、运算和输入/输出

### 1. 数组的赋值

在 CM Fortran 中,对数组的赋值提供了多种不同的方式。这里仅就数组构造符赋值、数组对数组的赋值和 DATA 语句对数组赋值作一简单介绍。

#### (1) 数组构造符赋值

数组构造符是专门为一维数组一次性地提供成批标量数组值。利用数组构造符,能方便地给一个数组赋初值。数组构造符提供了两种表示形式,可用其中任一种数组构造符给数组赋值均可。

两种数组构造符的书写形式为:

- [数组构造符值表]
- (/数组构造符值表/)

其中:数组构造符值表有五种形式:

①数组构造符值表是一个标量表达式,其特征是标量表达式由单个元素所构成的数组元素值。这种形式的赋值表示如下:

$$A = (/10,1,2,4,8,16,32,64,128/)$$

$B = [1, 7, -2, 333, -48]$

$C = [(0., 1.), 1, (2, 0)]$

②数组构造符值表是向量整型表达式,其特征是向量可以显示列出,也可以是前面已赋值的向量。这种形式的赋值可表示为:

$A = [V, [256, 512, 1024]]$  ! V 是一个向量表达式

$B = ([/0, 1, 2, 4, 8], [16, 32, 64, 128])$

③数组构造符值表是三元组式的隐循环数值表。其特征是,数组元素按照一定跨度组成整数顺序表。这种形式的赋值表示为:

$A = [1:1000]$

$B = (/2:100:2)$  ! 2 到 100 的偶整数向量

$C = [[1:100:2], [101:200:2]]$  ! 1 到 199 的奇整数向量

④数组构造符值表是由重复的数组构造符值表组成。这种形式可表示为:

$A = [100[5 * 10]]$  ! 100 个值是 50 的整型向量

$B = [100[1:100:2]]$  ! 100 倍个从 1 到 99 的奇整数型向量

⑤数组构造符值表是一个隐含的循环表,其特征是数组元素可以是整型、实型、复型和逻辑型。这种形式可表示为:

$A = (/(\sqrt{\text{real}(I)}), I=1:100:2))$

## (2) 数组对数组的赋值

数组对数组的赋值形式与一般变量赋值相似,赋值语句一般形式为:

数组名 = 数组表达式

在这里等式左边数组名可以是一个数组或数组段,等式右边的数组表达式是一个允许出现数组名或数组段的表达式。数组赋值必须满足两个条件:①等式左边的数组名与等式右边的数组表达式必须有相同的形状,所谓数组形状相同是指维数相同,每维长相同,但每维上下界不必相同;②等式右边的数组表达必须是一个标量。若不满足上述两个条件,则不能对数组赋值。例如,假设 A 和 B 都是一个  $10 \times 10$  的数组,当数组 A 形状为  $(2, 4, 5, 8)$ ,B 的形状为  $(3, 5, 1, 4)$  时,它们都是三行四列且有相同形状,则下述赋值语句是合法的。

$A(2:4, 5:8) = B(3:5, 1:4)$

执行上述语句后,不考虑 A 数组元素具体下标与 B 数组元素下标如何不同,由于它们的形状相同,总是 B 中的相应位置上的元素值赋给 A 的相应位置上。但是下列赋值语句是非法的:

$A(1:4, 1:3) = B(1:2, 1:6)$

因为即使在赋值语句的两边的数组中都有 12 个元素,但左边的形状是四行三列,而右边的形状则是二行六列,它们形状不同,所以是错误的。

当把一个标量赋值给一个数组时,标量的值则赋给该数组的所有元素。例如:若 C 是  $8 \times 9$  的数组,有赋值语句:  $C = 9$ ,则数组 C 中的 72 个数组元素的值均为 9。

## (3) 利用 DATA 语句对数组赋值

这里,我们仅仅介绍 DATA 语句中隐含循环赋值和带类型的 DATA 赋值两种:

①DATA 中带隐含循环的赋值形式为:

$\text{DATA } (\text{list}, i=m_1, m_2, m_3) / \text{数组元素个数} * \text{数组元素值} /$

其中: list 由数组元素和隐含 DO 表构成; i 是一个整变量名,  $m_1, m_2, m_3$  均是整常数表达式。

功能: 指定了由隐含 DO 循环执行后而得到的数组元素的表, 当它出现在 DATA 语句中时, 就必须有同样的个数的常数对数组中的元素依次赋值。

例如: 假定有  $4 \times 4$  的二维数组 A, 若有下列 DATA 语句时:

DATA((A(I,J),I=1,J),J=1,4)/10 \* 5/

则数组 A 中的上三角元素值均为 5, 可表示如下:

$$A = \begin{bmatrix} 5 & 5 & 5 & 5 \\ & 5 & 5 & 5 \\ & & 5 & 5 \\ & & & 5 \end{bmatrix}$$

在这里, A 中间空的元素表示没有赋初值。

又如 B 数组的形状是 [100, 100], 若有下列两个 DATA 语句:

DATA((B(I,J),J=1,I),I=1,100)/5050 \* 0.0/

DATA((B(I,J),J=I+1,100),I=1,99)/4950 \* 9/

则数组 B 的初值是: 下三角全置零, 上三角均置 9。

## ② 带类型说明的 DATA 语句的赋值

此种赋值形式, 是在数组说明时, 直接对数组赋给初值。赋值形式如下:

INTEGER, ARRAY(3), DATA::DRDER=[1, 2, 3]

REAL, ARRAY(4), DATA::D=[1.0, 2.0, 3.0, 4.0]

LOGICAL, ARRAY(3), DATA::MASK=[.false., .true., .false.]

利用类型说明语句直接对数组赋值的其它形式仅举例说明如下:

INTEGER A(0:9)=[10:19]

A(3:7)=-A(5:9)

执行上述两个赋值语句后, 数组 A 中的值是 [10, 11, 12, -15, -16, -17, -18, -19, 18, 19]。

## 2. 数组的运算

在数组运算表达式中允许使用算术操作符, 如 +, -, \*, /, \*\*, 允许使用操作数是数组名, 数组段, 数组构造符或常数。凡是参加运算的数组, 必须有相同的形状。

当数组与数组运算时, 两个数组作算术操作的结果仍然是一个形状相同的数组, 它的每个位置上元素的值是参与操作的相同位置上对应元素操作后的结果值。例如数组 A, 数组 B 和数组 C 都是  $3 \times 4$  的二维数组:

$$A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix} \quad B = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$$

若执行赋值语句:

C=A+B

则 C 的值就是 A 和 B 中对应的元素相加的结果。

$$C = \begin{bmatrix} 22 & 24 & 26 & 28 \\ 42 & 44 & 46 & 48 \\ 62 & 64 & 66 & 68 \end{bmatrix}$$

当数组表达式带有常数项时,则把常数项的形状看成与参加运算的数组形状相同。对于上述数组,若有下列语句:

$$C = A + B + 9$$

则常数 9 就看成是  $3 \times 4$  的二维数组中,每个元素均为 9,然后对应元素相加,结果值送给数组 C。可表示为:

$$C = \begin{bmatrix} 11+11+9 & 12+12+9 & 13+13+9 & 14+14+9 \\ 21+21+9 & 22+22+9 & 23+23+9 & 24+24+9 \\ 31+31+9 & 32+32+9 & 33+33+9 & 34+34+9 \end{bmatrix}$$

又如:假定 A,B 和 D 是  $10 \times 10$  的矩阵,F 是  $10 \times 100 \times 100$  的数组,S 是标量。下列数组表达式的赋值是合法的:

$$D = A * B + S + F(:, 1:10, 3)$$

### 3. 数组的输入/输出

数组的输入/输出与变量的输入/输出均相同,只不过对数组名和数组段输入/输出是连续的一组数值。对于输入/输出语句中既有数组名又有变量名时,按照排列的先后顺序,分别输入(输出)数组值和变量值。

若是一维数组,输入/输出比较简单,按照数组下标顺序输入(输出)。如下列输入语句:

READ \*, A, B(8:9), C(6)

当 A 数组的大小为 3 时,输入的数据应是 6 个,6 个值依次赋给 A(1),A(2),A(3),B(8),B(9)和 C(6)。

对于二维数组的输入/输出,需要注意的是 Fortran 自然存储按列优先存储,数组的数据输入和输出均与数学表示不一致。为了与数学表示一致,对二维数组的输入(输出)均使用隐含 DO 表,以免造成不必要的错误。隐含表中将列变量作为内循环,行变量作为外循环,这样输入(输出)就与数学表示完全一样。假设 A 是  $4 \times 5$  的二维数组,输入语句表示如下:

READ \*, ((A(I,J), J=1,4), I=1,5)

执行时应该读入 20 个数据,依次赋给 A(1,1),A(1,2),A(1,3),A(1,4),A(2,1),A(2,2),...,A(4,5)。这样就完全与数学表示一致,输出同样如此。

### 4. 编程举例

#### 例 5.5 用高斯-赛德尔迭代法解方程组。

若方程组  $\sum_{j=1}^n a_{ij}x_j = b_i, i=1, 2, \dots, n$  的系统矩阵具有主对角线优势,即满足下列条件:

$$\sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| < |a_{ii}|, \quad i=1, 2, \dots, n$$

则高斯-赛德尔迭代式为:

$$x_i^{(k+1)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right) / a_{ii}, \quad i=1, 2, \dots, n$$

设初值  $x_i^{(0)} = 0, \quad i=1, 2, \dots, n$

结束迭代的条件为:

$$\max_{1 \leq i \leq n} |x_i^{(k+1)} - x_i^{(k)}| < \epsilon$$

其中,  $\epsilon$  为给定的精度要求。即后一次迭代解比前一次迭代解之差的绝对值都分别小于  $\epsilon$  时, 则这最后一次迭代的  $x_i$  值就是所求的解。在本程序中取  $N=10$ 。程序可为:

```
PROGRAM Gauss_seidel_method
    IMPLICIT NONE
    INTEGER I,J,K,MAX
    REAL SUM,R,T,E
    REAL ARRAY(10)::A(10,10),B,x
    E=0.00001;MAX=10;x=0.0
    PRINT *, 'INPUT COEFFICEN ARRAY A:'
    READ *, ((A(I,J),J=1,10),I=1,10)
    PRINT *, 'INPUT COEFFICEN ARRAY B:'
    READ *, B
    Loop-K:      DO K=1,MAX
        R=0.0
        Loop-I:      DO I=1,10
            SUM=0.0
            Loop-J:      DO J=1,10
                SUM=SUM+A(I,J)*x(J)
            END DO Loop-J
            T=(B(I)-SUM)/A(I,I)
            IF(ABS(T-x(I)).GT.R)R=ABS(T-x(I))
            x(I)=T
        END DO Loop-I
        WRITE *, x(K)
        IF(R.LT.E) EXIT
    End
```

```

INTEGER ARRAY(50)::score,name
INTEGER I,J,N,SC,NA
REAL SUM,AVERAGE
SUM=0.0
PRINT *,'READ N'
READ *,N
PRINT *,'INPUT score AND name'
DO I=1,N
    READ *,score(I),name(I)
    SUM=SUM+score(I)
END DO
SORT:DO I=N,2,-1
Loop-J: DO J=1,I-1
    IF (score(J).GT.score(J+1) THEN
        SC=score(J);na=name (J)
        score(J)=score(J+1);name(J)=name(J+1)
        score(J+1)=score(J+1);name(J)=name(J+1)
        score(J+1)=SC;name(J+1)=na
    end if
    END DO Loop-J
END DO SORT
AVERAGE=SUM/N
PRINT *,'OUT SORT SCOREN AND NAME'
DO I=N,1,-1
    PRINT *,N-(I-1),score(I),name(I)
END DO
PRINT *,'AVERAGE SCORE=',AVERAGE
END PROGAM SCORE-SORT

```

### § 5.3 不同形式的数组说明

一个数组的形式和维界，由一个数组说明语句来决定。数组的形式说明可以出现在类型说明语句中，也可以出现在 DIMENSION 语句中，或者出现在一个公用语句中进行数组说明。

例如：下列两种说明的数组是等价的。

①DIMENSION A(2:5,2:6),B(8,9)

    INTEGER A,B

②INTEGER A(2:5,2:6),B(8,9)

说明数组的形式有四种：显示形式数组说明，假定形式数组说明，延迟形式数组说明，假定大小数组的说明。

显示形式数组能说明全局的(在一个公用块中)或局部的数组;假定形式数组和假定大小数组只能说明哑元过程的数组;延迟形式数组能说明全局的、局部的或哑元的数组。

- 显式形式数组说明中规定的维界是每个数组的大小;全局(公用)数组的维界必须是常量;而局部数组的维界可以在执行过程时来固定。

- 假定形式数组说明中规定一哑元数组的排列,但每个数组维数的大小是在执行数组变元的过程中才能确定。

- 延迟形式数组说明中规定了一个数组分配的排列方式,数组的范围大小,只有当指定该数组进行分配时才能确定。

- 假定大小数组说明中规定一个哑元数组排列以及除最后一维外的其它数组维界的大 小。而数组中的最后一个维界大小,必须从程序中来确定。

上述四种数组说明是由数组的语法规定的,它们的语法规定如下。

### 1. 显式形式数组说明

一个显式形式数组说明中,规定了数组的每个维界表达式都是整常数表达式。即维界表达式里只能出现整常数或整常数名,不得出现整型变量或其它的操作数。每个显式形式数组说明中明确规定了显式形式数组的维数和形状的大小。显式形式数组的应用范围最广,它可以用于主程序中,也可以用于子程序中;它可以是 CM 数组,也可以是一个前端处理机的数组。

一个显式形式数组说明的一般形式为:

类型说明 数组名(显式形式说明维界[, 显式形式说明维界]...)

其中:①显式形式说明维界是指上界:下界。

②显式形式说明维界的最大个数是七个。

每一个下界和上界的值确定数组维数的使用范围。

显式形式数组说明举例如下:

```
REAL A(N:N+99),B(0:N-1),C(10,100,N * N)
INTEGER D(-1000:2000,N+1),E(N,N,N,N,N,N,N)
```

其中:表达式中的 N 是整常数值的符号名。

第一个显式形式数组说明了三个实型数组,A 和 B 为二维实型数组,C 为三维的实型数组。第二个显式形式数组说明了二个整型数组,D 为二维整型数组,E 为七维整型数组。

### 2. 假定形式数组说明

一个假定形式数组是一个哑元数组,它没有显式形式说明的上界(或没有显式说明的上下维界),只有当执行过程时作用在变元数组时才加以说明确定。一个假定形式数组是由类型说明(或 DIMENSION 语句)、哑元数组名和假定形式数组说明所组成。一个假定形式数组总是一个 CM 数组。

一种假定形式数组说明,每维都没有上下维界,只用“:”表示。它的一般形式为:

类型说明 数组名(假定形式说明[, 假定形式说明]...)

其中:假定形式说明由“:”来表示。

一个假定形式数组的维数等于假定形式数组说明时所用符号“:”的个数。假定形式数组的维数说明不得超过七个。

一种假定形式数组说明的例子是:

```
REAL A(:,B(:,:)
INTEGER D(:,:,:,:,:,:)
```

第一个假定形式数组说明了二个实型数组,A 为一维的假定形式数组,B 为二维的假定形式数组;第二个假定形式数组说明了一个整型数组,D 为七维假定形式数组。

一个假定形式数组不能在主程序中使用与定义,只能在函数子程序或子例子程序中作为哑元数组使用。显然,当一个假定形式数组在子程序中说明时,它没有确定的上下维界,只有在主程序调用该子程序作哑实结合时,就取相结合的实元数组上下维界作假定形式数组的上下维界。假定在一个主程序中有一个实元数组的形状为(-3:8,1:9)与假定形式数组 B 相结合,此时 B 的实际维界也是(-3:8,1:9)。

另一种假定形式数组说明,数组中的每个维都有显式形式的下界,而没有确定的上界。例如:

```
REAL A(N:),B(0:,0:)
INTEGER C(10:,100,N*N:),D(-1000,N+1:)
```

其中:N 是一个事先预置的常量或是一个哑元变量或是一个公用变量。

第一个假定形式数组说明了二个实型数组,A 为一维的假定形式数组,B 为二维的假定形式数组;第二个假定形式数组说明了二个整型数组,C 为三维数组,D 为二维数组,使用范围与前一种相同,它的上界也是由调用它的哑实结合时实元数组所确定,而此时假定形式数组的上界实际取值要确保该数组中的每个维长与实元数组的维长相等。即假定形式数组调用实元数组的维长是相等的,但它们的上下界值不一定相同。例如:假设调用它的一个实元数组形式为(-3:8,1:9)与假定形式数组 B 相结合,此时 B 的实际维界是(0,11,0,8)。

下面举例说明一个假定形式数组的应用。子程序 INIT2INT 是对一个两维整型数组进行初始化,而子程序 PRINT2INT 是打印一个二维整型数组。

```
SUBROUTINE INIT2INT(A)
INTEGER A(:, :)
INTEGER U(2)
U=DUBOUND(A)
FORALL (I=1:U(1),J=1:U(2))A(I,J)=I * 10 + J
END
```

```
SUBROUTINE PRINT2INT(A)
INTEGER A(:, :)
```

...

! 此段程序是打印数组 A 的值  
END

PROGRAM TEST

```
INTEGER A(4,7),B(2,9)
CALL INIT2INT(A)
CALL PRINT2INT(A)
END
```

### 3. 延迟形式数组的说明

一个延迟的形式数组是一个可分配的数组,或者是数组指针对延迟形式数组的维数说明。但是这延迟形式数组的大小范围是不能确定的,直到指定的动态分配给该数组的一个 ALLOCATE 语句执行时为止,此时由 ALLOCATE 语句将延迟形式数组的维界确定下来。延迟形式数组的维界在程序的执行过程中可随时按需要进行变化,数组需要占多少内存,就可在程序中动态地分配给数组多少内存;如果该数组以后不再使用,又可调用 DEALLOCATE 语句将该数组所占内存进行释放,归还给系统另作它用。这样可以节约内存的使用,提高内存的使用效率。当一个数组是执行数据的输入或者数据的求值时,而且该数组的维界直到运行时都是未知的,此时延迟形式数组格外有用,程序随时对延迟形式数组进行分配(ALLOCATE)和释放(DEALLOCATE),存储器的内存管理,直接由程序来控制。延迟形式数组可用主程序和子程序中。延迟形式数组的说明与假定形式数组的说明相似,也用冒号“:”代表数组的维数,但不同的是延迟形式数组的说明中必须包括属性说明 ALLOCATABLE 语句或 POINTER 语句。

延迟形式数组说明的一般形式为:

类型说明,属性说明::数组名(:[,,:])…)

延迟形式数组的维数等于延迟形式数组说明时的冒号“:”数,但最多维数不得超过七维。

例如: 下列信息延迟形式数组说明的例子:

```
REAL,ALLOCATABLE::D(:, :, :),E(:, :, :, :)
INTEGER,POINTER::R(:, :, :)
```

第一个延迟形式数组说明两个可分配实型数组 D 和 E,维数分别是二维和三维数组。其中属性 ALLOCATABLE 意为“可分配”,即该数组属于可分配内存的延迟形式数组。

第二个延迟形式数组说明一个指针型的整型数组 R,维数是二维数组。其中属性 POINTER 意为“指针”,即该数组属于指针型的延迟形式数组。

延迟形式数组说明中的属性 ALLOCATABLE 和属性 POINTER 是延迟形式数组的主要特征,有别于假定形式数组的说明。尽管假定形式数组说明和延迟形式数组说明的数组形式都用冒号“:”表示(即(:, :, :)),而有属性 ALLOCATABLE 和属性 POINTER 的数组是延迟形式数组,数组所占内存是动态的,主要用于主程序和子程序。而无属性说明的数组是假定形式数组,数组从哑元结合时起占据连接机内存空间,只有当程序段结束时,内存才被释放。假定形式数组说明的数组,总是一个连接机中的数组。

#### 4. 假定大小数组说明

一个假定大小数组是一个哑元数组,该数组的最后一维上界是未知的。一般用星号来表示(或缺省)。未知的上界必须假定在程序的执行过程中,由其它哑元变量或通过公用变量所提供。当数组说明语句中说明一个数组的最后一维上界是未知时,这样的数组说明称为假定大小数组说明。

假定大小数组说明的数组,只能用于函数或子例子程序,而且假定大小数组必须作为哑元出现在 SUBROUTINE 或 FUNCTION 语句的哑元表中。一个实际数组与哑元数组的排列和范围可以不同,但实际数组最后大小必须是哑元数组的说明过程中所假定的大小。一个假定大小的数组,总是一个前端处理机的数组。

假定大小数组说明举例如下:

```
DIMENSION A(N:N+99,1: * ),B(10,100,N * N,10: * )
```

```
DIMENSION C(-1000:2000,N+1,*),D(1:50,5,*)
DIMENSION E(0:N-1,0:*,E(N,N,N,N,N,N,*))
```

其中:N 是一个常量名,或是一个哑元变量,或是一个公用变量;凡是未给出某个维数的下界,系统均设定下界值是1。

假定大小数组的体积,完全依赖于与它相结合的实元。若实元为数组名,则假定大小数组的体积与实元相同。若实元为数组元素名,则假定大小数组的体积等于实元数组从该数组元素起到实元数组最后一个元素为止的那部分体积。若实元是字符数组名,或字符数组元素名,或字符数组元素子串名,则假定大小数组的体积(以字节计算)等于实元数组从第一个元素的第一个字节,或数组元素的第一个字节,或数组元素子串的第一个字节起到实元数组最后一个元素的最后一个字节为止的那部分体积。

假定大小数组除最后一维以外的那部分体积不得大于上述的实元体积。

## § 5.4 数组的屏蔽

在 CM Fortran 的数组运算操作中,专门提供了一种功能很强的数组屏蔽语句(WHERE 语句)和数组屏蔽结构(WHERE 结构)。一个数组屏蔽语句,只对屏蔽数组中数组元素为“真”的对应位置进行赋值和运算操作。一个数组屏蔽结构是供数组作选择分叉使用,它不仅能对逻辑条件为“真”的条件下完成数组运行操作,同时能对 ELSEWHERE 语句在逻辑条件为“假”的条件下实现对数组的运算操作。

### 1. WHERE 语句

WHERE 语句的一般形式为:

WHERE(数组关系表达式)数组赋值语句

其中数组关系表达式是以数组名为操作对象的关系表达式,当数组关系表达式中逻辑条件为真时(或数组关系表达式是一个逻辑数组且逻辑数组中元素为真的位置),才对应的执行数组的赋值语句,否则不执行数组赋值语句。

**例 5.7** 设 A,B 是  $10 \times 10$  的数组,若有如下的 WHERE 语句:

WHERE (B.NE.0)A=A/B

则只要 B 中有不等于零的元素,就可以对应执行除法运算,然后赋给 A 中对应的位置上。假定数组 B 中主对角线之上(包括主对角线)所有元素均不为零,主对角线之下全为零,执行上述语句,计算机将并行把 B 中不为零的元素对应的去除 A 中的元素,除得的结果并行地送到 A 中对应的位置上,A 中其它元素不变。上述赋值语句相当于下列程序段:

```
Loop—I:DO I=1,10
Loop—J:    DO J=1,10
            IF B(I,J).NE.0 THEN A(I,J)=A(I,J)/B(I,J)
            END DO Loop—J
        END DO Loop—I
```

例如:假定 x,y,z 都是一维数组,若执行下列 WHERE 语句时:

WHERE (x(N:N+59).LE.100) y(1:60)=SUM(z(1:100))

当 N 取不同的值时,一维数组 x 中数组元素就有所不同,但有一点是不变的。不管 N 取什么

值,对于数组 X 总是取连续的 60 个元素进行判断。假若选取 60 个数组元素值均小于等于 100 时,就将数组 Z 中前 60 个数组元素值逐项累加求和并分别送到对应的数 Y 中。

**例 5.8 假定 DATA, TEMP, HISTO 均为一维数组,执行下列一段程序:**

```
DO I=MIN,MAX,STEP
    WHERE (I .LE. DATA .AND. DATA .LT. I+STEP) TEMP=DATA
    HISTO=SUM(TEMP,I)
    PRINT *,I,I+STEP,HISTO
END DO
```

此处 WHERE 语句,从数组 DATA 中循环地选择数组元素值送到数组 TEMP 中的对应元素位置上。接着下一个语句是对数组 TEMP 进行求和(这里数组 TEMP 中的数组元素值在循环过程中将有所改变),将结果存入数组 HISTO 中,然后打印出来。

## 2. WHERE 结构

WHERE 结构有三种表示形式:

### ①最简单的 WHERE 结构

```
WHERE(屏蔽表达式)
    WHERE 结构体语句
END WHERE
```

其中:WHERE 结构体语句可以包含三种情况:第一种是数组的赋值语句;第二是 WHERE 语句;第三是 WHERE 结构(即 WHERE 结构体语句又包含有 WHERE 结构)。

功能:当屏蔽表达式的逻辑值为真时,执行 WHERE 结构体中满足条件的那部分语句。

例如: WHERE (PRESSURE .LT. 1.0)

```
PRESSURE=PRESSURE+INC+PRESSURE
TEMP=TEMP-5
END WHERE
```

其中,PRESSURE,INC,PRESSURE,均为相同形状的数组。

### ②带有 ELSEWHERE 语句的结构

```
WHERE(屏蔽表达式)
    WHERE 结构体 1
ELSEWHERE
    WHERE 结构体 2
END WHERE
```

功能:在该 WHERE 结构中,当屏蔽表达式的逻辑值为真的情况下,对 WHERE 结构体 1 中的满足条件的数组元素进行赋值,同时也能对 ELSEWHERE 的后续语句 WHERE 结构体 2 中那些不满足逻辑条件的数组元素进行赋值。

例如:假定 A,B,C 均是相同形状的数组,若执行下列程序段:

```
WHERE (A<>0)
    C=B/A
ELSEWHERE
    C=0
```

---

```
END WHERE
```

对数组 A 中那些不为零的数组元素去除数组 B 中对应元素, 然后将结果送到数组 C 中对应元素的位置上。同时对数组 A 为零的那些元素位置所对应数组 C 中的元素位置全部置零。

### ③带有 ELSEWHERE(屏蔽表达式)语句的结构

```
WHERE (屏蔽表达式 1)
```

```
    WHERE 结构体 1
```

```
ELSEWHERE (屏蔽表达式 2)
```

```
    WHERE 结构体 2
```

```
ENDWHERE
```

此种 WHERE 结构与第二种结构形式的区别在于只有屏蔽表达式 2 中的逻辑条件为真的情况下, WHERE 结构体 2 中满足条件的数组元素才被执行。在这里, 屏蔽表达式 1 和屏蔽表达式 2 可以是对同一个数组中的元素进行屏蔽, 也可以是对不同的数组元素进行屏蔽。

例如: 假定  $x, y, z$  均为相同形状的数组, 且:

$$x = \begin{bmatrix} -1 & 2 & 0 \\ 4 & -2 & 5 \\ 0 & 7 & -3 \end{bmatrix}, \quad y = \begin{bmatrix} 11 & 12 & 8 \\ 20 & 22 & 45 \\ 9 & 28 & 33 \end{bmatrix}, \quad z = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

若执行下列程序段:

```
WHERE ( $x > 0$ )
```

```
     $z = y/x$ 
```

```
ELSEWHERE ( $x < 0$ )
```

```
     $z = x$ 
```

```
END WHERE
```

则数组  $z$  的结果为:  $z = \begin{bmatrix} -1 & 6 & 0 \\ 5 & -2 & 9 \\ 0 & 4 & -3 \end{bmatrix}$

即: 当数组  $x$  中元素值大于零时, 就将该数组元素值去除  $y$  中的对应元素值, 结果赋给数组  $z$  中的对应元素的位置上。当数组  $x$  中元素值小于零时, 就将该数组元素值直接赋给数组  $z$  中对应的位置上。数组  $z$  中的其余元素不变。

### 3. WHERE 结构块在多处理机中的控制作用

在并行计算中存在着两种执行方式, SIMD 和 MIMD。MIMD 能较好地独立地执行分支和转移, 但同步、通讯却不方便; SIMD 较好地解决了同步和通讯问题, 但执行分支转移的效率低下。而 CM Fortran 综合了 SIMD 和 MIMD 的优点, 克服了 MIMD 在同步、通讯中的不方便, 解决了 SIMD 在执行分支转移时效率低下的问题。例如下列程序段:

```
DIMENSION A(100), B(100), C(100)
```

```
...
```

```

WHERE (B+C .GT. 100)
  A=B+C
ELSEWHERE
  A=B*C
ENDWHERE

```

为了讨论问题的方便,假定 WHERE 结构块执行时处理机个数 $\geq 100$  同时空闲。当前 50 个元素满足 WHERE 的条件,即  $B(1:50)+C(1:50) > 100$  执行 WHERE 结构块。如果假定按 SIMD 方式执行,则首先前 50 个处理机执行  $A(1:5)=B(1:50)+C(1:50)$ ,另外后 50 个处理机处于等待状态。当前 50 个处理机执行完毕时,后 50 个处理机才执行  $A(51:100)=B(51:100)*C(51:100)$ ,这时先前的 50 个处理机处于等待状态。只有当后 50 个处理机执行完毕后,100 个处理机才在 ENDWHERE 处汇聚,再往下执行。

按 MIMD 方式执行 WHERE 结构块时,有 50 个处理机执行  $A(1:50)=B(1:50)+C(1:50)$ ,同时另 50 个处理机执行  $A(51:100)=B(51:100)*C(51:100)$ ,谁先执行完谁就接着执行 ENDWHERE 之后语句,显然 MIMD 执行的效率比 SIMD 要高,但因为数据相关关系,程序中必须加入同步指令,程序调试中可能会出现错误不可再现,使得 MIMD 程序难于理解和调试。而采用 CM Fortran 程序执行时,既按 MIMD 方式执行又按 SIMD 方式在 ENDWHERE 处汇聚,既可得到 MIMD 的高效率,又可以使 SIMD 的程序流程控制简单,程序易于调试。

#### 4. WHERE 结构的嵌套

WHERE 结构只能用于数组操作,并对数组中的元素有选择地进行操作。一个屏蔽表达式限定了 WHERE 结构体内的数组表达式的求值操作和对数组元素的赋值操作。一个 WHERE 结构体内能包含如下操作:

- i) 一个 WHERE 的赋值;
- ii) 一个 WHERE 的语句;
- iii) 其它的 WHERE 结构。

也就是说,若 WHERE 结构体内还包含有另外的 WHERE 结构,这种包含关系就称为嵌套的 WHERE 结构。即在 CM Fortran 中,不仅提供了 ELSEWHERE 语句,使得数组元素在不同的子集中能同时进行并行运算和并行赋值操作,而且还提供了嵌套的 WHERE 结构,使得并行程序设计具有更大的灵活性和适应性。

下面是一些相同问题,使用嵌套结构表示和不使用嵌套结构形式的对比描述。

① 使用嵌套结构描述的第一种形式: 不使用嵌套结构描述上述形式为:

WHERE (屏蔽表达式 1)

赋值语句

ELSEWHERE

WHERE (屏蔽表达式 2)

赋值语句

END WHERE

END WHERE

WHERE (屏蔽表达式 1)

赋值语句

ELSEWHERE (屏蔽表达式 2)

赋值语句

ENDWHERE

## ② 使用嵌套结构描述的第二种形式:

WHERE (屏蔽表达式 1)

赋值语句

WHERE (屏蔽表达式 2)

赋值语句

END WHERE

赋值语句

END WHERE

也可以将上述嵌套结构改变成如下形式:

不用嵌套结构描述上述形式为:

WHERE (屏蔽表达式 1)

赋值语句

END WHERE

WHERE (屏蔽表达式 1 . AND.

屏蔽表达式 2)

赋值语句

END WHERE

WHERE (屏蔽表达式 1)

赋值语句

END WHERE

TEMP1=屏蔽表达式 1

WHERE (TEMP1)

赋值语句

END WHERE

TEMP2=屏蔽表达式 2

WHERE (TEMP1 . AND. TEMP2)

赋值语句

END WHERE

WHERE (TEMP1)

赋值语句

END WHERE

## ③ 使用嵌套结构描述的第三种形式:

WHERE (屏蔽表达式 1)

WHERE (屏蔽表达式 2)

赋值语句 1

ELSEWHERE (屏蔽表达式 3)

赋值语句 2

END WHERE

ELSE WHERE

赋值语句 3

END WHERE

也可将上述嵌套结构描述如下形式:

WHERE (屏蔽表达式 1 . AND. 屏蔽表达式 2)

赋值语句 1

WHERE (屏蔽表达式 1 . AND. 屏蔽表达式 3)

赋值语句 2

END WHERE

```
ELSEWHERE
```

赋值语句 3

```
END WHERE
```

综上所述,对于多种屏蔽条件的复杂问题,用 CM Fortran 中的嵌套结构来设计程序,能使程序变得简单,易读,易懂。

## § 5.5 数组元素的分配语句 FORALL

FORALL 语句是 CM Fortran 在 Fortran 90 之上新增加的扩充语句,它是数组元素分配过程中最能体现现代特征的语句之一。CM Fortran 能并行地实现某些数据集(或数据子集)的赋值操作。用 FORALL 语句构造的程序,能明显地减少顺序限制,从而实现优化程序的执行时间。FORALL 语句的语义与 DO 循环结构相似,但赋值操作是并行执行的。

### 1. FORALL 语句结构形式

FORALL 语句的一般结构形式为:

FORALL(FORALL 三元组说明表[,标量屏蔽表达式])FORALL 赋值语句

其中:①FORALL 三元组说明表中,可以同时说明多个三元组下标变量。而任一个三元组下标变量的说明形式均为:

下标变量=下标变量下界:下标变量上界[:下标变量的跨度(或增量)]

②标量屏蔽表达式是可选项,若 FORALL 语句中带有可选项,则只有标量屏蔽表达式中的逻辑条件值为真的那些数组元素的子集,才能实现对应的数组元素的赋值操作。

③FORALL 赋值语句,是对数组元素满足一定的条件进行赋值操作。

功能:该语句对数组元素集并行地执行赋值操作。

### 2. FORALL 语句中的一般赋值操作

此种赋值操作,是指不带屏蔽限制条件的赋值操作。以下举例说明。

**例 5.9** 给  $4 \times 3$  的数组 A 进行初始化

FORALL (I=1:M,J=1:N)A(I,J)=I+(J-1)\*M

则数组 A 的结果是:

$$A = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}$$

**例 5.10** 对例 5.9 中的数组 A 的元素进行转置,

FORALL (I=1:M,J=1:N)A(I,J)=A(J,I)

则转置的数组 A 的结果为:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

**例 5.11** 利用前缀加法,使得数组 B 中的每个元素是数组 A 中相应元素之前的所有元素之和。

```
FORALL (I=1:N) B(I)=SUM(A(1:I))
```

结果(假设 A 中元素值为 1~10)可表示为:

A(I)	1	2	3	4	5	6	7	8	9	10
B(I)	1	3	6	10	15	21	28	36	45	55

在 CM Fortran 中,FORALL 语句能对全体数组或部分数组中的个别元素进行赋值操作。

例如:将向量 V 沿着二维矩阵 H 中的第一维进行展开。

```
DIMENSION H(N,M),V(M)
```

...

```
FORALL (I=1:N)H(I,:)=V
```

执行 FORALL 语句之后,将向量 V 中的值,分配到矩阵 H 中的每行之中。即矩阵中的每行元素值均与向量 V 中的值相同。

### 3. 条件 FORALL 语句

在一个 FORALL 语句中若包含了屏蔽表达式,则表示对数组元素进行有选择地赋值操作。条件 FORALL 语句的效果类似于 DO 结构体内包含有一个条件语句。屏蔽表达式总是一个标量值的表达式,对数组中部分的元素进行屏蔽。

**例 5.12** 在 FORALL 语句中为了避免赋值表达式中除数为零的情况,即可设置一个屏蔽表达式,排除数组元素中值为零的那些数组元素:

```
FORALL (I=1:N,A(I).NE.0.0)B(I)=1.0/A(I)
```

在这里,数组 A 和数组 B 是相同形状的数组。

**例 5.13** 用条件 FORALL 语句屏蔽矩阵 H 中的部分元素:

```
FORALL (I=1:N,J=1:N,I.GT.J)H(I,J)=0.0
```

结果正方形矩阵 H 的对角线下方清为零。

**例 5.14** 下面是用五点差分格式求解 Laplace 方程的过程中,采用红-黑迭代方案时,对数组进行初始化的程序段。其中 F 为数值型数组,amask 和 bmask 为逻辑型数组。

```
amask=.false. ;bmask=.false.
```

```
FORALL (i=2:x-1,j=2:y-1,mod(i+j,2).EQ.0)amask=.true.
```

```
FORALL (i=2:x-1,j=2:y-1,mod(i+j,2).EQ.1)bmask=.true.
```

```
F=2.0;F(:,y)=1.0
```

```
WHERE (amask)F=0.0
```

```
WHERE (bmask)F=0.0
```

### 4. 用 FORALL 语句实现数据的移动

FORALL 语句不仅具有很强的数据移动功能的特点,而且还具有摹拟众多 CM Fortran 中实现数组变换的特殊函数等特点(数组变换在下一章介绍)。FORALL 语句并行地实现数据移动的功能,很难用 CM Fortran 其它语句来并行地实现。

**例 5.15** FORALL 语句能并行地执行任意多维数组的排列与置换。下列语句中将数组 x 和数组 y 作为矩阵 H 的下标进行赋值操作:

```
FORALL (I=1:N, J=1:N) G(I,J)=H(x(I,J), Y(I,J))
```

FORALL 语句也能对不规则形状数组部分进行并行的操作。如将矩阵 H 中对角线元素抽出并赋给向量 V:

```
FORALL (I=1:N) V(I)=H(I,I)
```

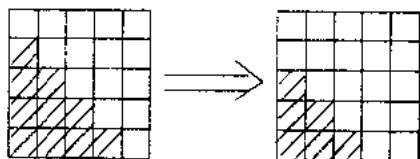
该语句执行后的结果表示如下图:



**例 5.16** 将一个下三角矩阵 H 进行并行地缩小:

```
FORALL (I=1:N, J=1:I-1) H(I,J)=H(I,J+1)
```

结果如下图所示:



**例 5.17** 下列各 FORALL 语句等价于一些数组变换的特殊函数,而 FORALL 语句比数组变换函数更灵活,适应范围更广。

①FORALL (I=1:M, J=1:N) A(I,J)=B(J,I) 是等价于下列数组的转置:

```
A=TRANSPOSE(B)
```

②FORALL (L=1:M, K=1:N) A(L,K)=SUM((L:,)\*B(:,K)) 是等价于矩阵  
A=MATMUL(A,B) 的乘法。

③FORALL (J=1:N) D(I,J)=V 是等价于下列数组构造函数(在这里,V 为数组或向量)。

```
D=SPREAD(ARRAY=V,DIM=2,NCPIES=N)
```

## § 5.6 动态分配

动态分配是通过分配(ALLOCATE)语句和释放(DEALLOCATE)语句,来支持管理那些可分配数组和指针数组(指针数组也简称为指针)的存储。

可分配数组是由属性 ALLOCATABLE(可分配)来说明的,而指针数组是由属性 POINTER(指针)来说明的。一个可分配数组或指针数组的大小,维界和形状是未定的,一直到该数组由 ALLOCATE(分配)语句执行分配为止。可分配数组和指针数组,它能像任何其它的 CM 数组一样使用。

指针数组不仅对于局部的程序单元能够提供像可分配数组同样的功能,而且指针数组还提供了动态管理全局存储的能力。并且指针数组能够出现在公用块中,而可分配数组却不能出现在公用块中。

### 1. 可分配数组的说明

在数组说明语句中,若带有属性 ALLOCATABLE(可分配)的关键字,则称该数组为可分配数组。一个可分配数组是一个动态数组,可分配数组中内存单元的使用,是由 ALLOCATE(分配)语句和 DEALLOCATE(释放)语句来进行动态的管理。

可分配数组说明的一般形式为:

类型说明,ALLOCATABLE::数组名[:,:,:]…[:,:,:]…

其中: ALLOCATABLE 为可分配数组属性的关键字。在该数组说明中,带有 ALLOCATABLE 属性的关键字的数组就是动态数组,没有带 ALLOCATABLE 属性关键字的数组就是延迟形式数组。数组的维数等于延迟形式数组中冒号“:”的个数。

一个可分配数组的大小,维界和形状是未定义的,直到由一个 ALLOCATE(分配)语句执行分配为止。一个可分配数组在未分配之前,该数组中的任何部分不能被引用。下面是可分配数组说明的例子:

**例 5.18**

```
REAL,ALLOCATABLE,ARRAY(:, :, :),A,B,C
INTEGER,ALLOCATABLE::D(:,:,:,:)
...
```

该例中的第一个说明语句说明三个数组 A,B,C 为二维的实型动态数组;第二个说明语句说明数组 D 为三维的整型动态数组。

**例 5.19**

```
REAL,A,B(:, :)
ALLOCATABLE::A(:, :, :),B
...
```

例 5.19 中的第一个说明语句是说明数组 A 和 B 为实型的数组,第二个说明语句进一步说明数组 A 和 B 是两个动态数组。

## 2. 指针数组的说明

一个数组在类型说明时,若给出了指针属性 POINTER 关键字,则称该数组为指针数组。一个指针数组是一个动态数组。指针数组中存储单元的使用是由 ALLOCATE(分配)语句和 DEALLOCATE(释放)语句来进行动态的管理。

指针数组说明的一般形式:

类型说明,POINTER::数组名[:,:,:]…[:,:,:]…

其中: POINTER 为指针属性的关键字。指针数组中的维数等于延迟形式数组中冒号“:”的个数。

一个指针数组的大小、维界和形状都是未定义的,只有当执行到一个 ALLOCATE(分配)语句时,该指针数组的大小、维界和形状才是确定的。一个指针数组在未分配之前,该数组中的任何部分不能被引用。下面是指出针数组说明的例子:

**例 5.20**

```
REAL,POINTER,ARRAY(:, :, :),x,y,z
或者:
```

```
REAL,ARRAY(:, :, :),x,y,z
POINTER x,y,z
```

此例说明了三个指针数组 x,y,z 均为一维实型的指针数组,尽管两种说明方式不一样,但它们的功能是等价的。

**例 5.21** 在下列例子中,说明三个指针数组 P,Q 和 R 的多种等价方法:

---

```
INTEGER,POINTER::P(:),Q(:,R(:,:))
```

或者：

```
INTEGER P,Q,R  
POINTER P(:),Q(:,R(:,:))
```

或者：

```
INTEGER,POINTER,ARRAY(:,P,Q  
INTEGER,POINTER,ARRAY(:,R)
```

或者：

```
INTEGER P,Q,R  
DIMENSION P(:,Q(:,R(:,:))  
POINTER P,Q,R
```

### 3. 分配状态

在程序开始执行时,一个可分配数组或指针数组的分配状态是处于非当前分配状态,并且可分配数组或指针数组不能是定义的。只有当一个程序执行到一个 ALLOCATE 语句后,这时可分配数组或指针数组才是实际上分配了内存,并能进行有效地使用。如果对非当前分配状态的可分配数组或指针数组进行使用,计算机将立即指出使用方法错误。

当用一个 DEALLOCATABLE 语句对一个已分配数组或指针数组进行释放内存,这时可分配数组或指针数组的状态,立即又变成了非当前分配状态,它又是一个未定义的可分配数组或指针数组。

如果一个已分配的数组或指针,用在一个过程中,当遇到 RETURN 语句或 END 语句终止时,此时的分配状态随过程结束而变成了未定义(即相当于执行了 DEALLOCATE 语句)。对于一个未定义分配状态的数组或指针,不能进行引用。

当要确定一个可分配数组是否为分配状态时,可由内部函数 ALLOCATED 来测定。若所测数组的函数值为“真”,则说明该数组处于已分配状态。否则,说明该数组处于未分配状态。

当要确定一个指针数组是否为分配状态时,可由内部函数 ASSOCIATED 来测定。若对所测指针数组的函数值为“真”时,则说明该指针数组处于未分配状态。

例如:假定 P 说明为指针数组,对 P 进行测定。若指针数组 P 为分配状态,显示“P 已经分配”,若测定指针数组 P 未分配,则用 ALLOCATE 语句对指针数组进行分配。下面是完成上述功能的程序段:

```
IF (ASSOCIATED(P) THEN  
    PRINT *, 'P is already allocated.'  
ELSE  
    ALLOCATE(P(0:M))  
END IF
```

### 4. ALLOCATE(分配)语句与 DEALLOCATE(释放)语句

ALLOCATE 语句又称分配语句,该语句的功能是按需要为动态数组分配各维的维界,使动态数组占有所需要量的内存。即:ALLOCATE 语句为可分配数组或指针数组动态地分配内存单元。

ALLOCATE 语句的一般形式为：

ALLOCATE(数组名 1(显式形状说明), 数组名 2(显式形状说明)…)

其中：显式形式说明格式为：[下界:]上界。在这里的上下界的值是确定值。一个 ALLOCATE 语句可以为多个动态数组分配维界，个数不限。但需要注意几点：

①每一个数组名必须是一个可分配数组，而且该可分配数组是一个非当前分配的数组名或指针数组名。

②此处的维界是确定的值，而不能是一个表达式。

③在一个分配数组名中的显式形式说明的维数必须与原可分配数组名中有同样的维数。例如：假定要为动态数组 A,B 分配维界，且 N,M 已有定义值，希望一维数组 A 分配形式为(1:N)，二维数组 B 分配形式为(-3:M,0:9)，只要在程序执行部分需分配具体维界处写上分配语句：

ALLOCATE (A(N),B(-3:M,0:9))

就可达到分配的目的。此时就相当于在程序说明中用类型说明语句说明 A 是一维数组且形式为(1:N),B 是二维数组且形式为(-3:M,0:9)，并且 N,M 已有确定值。

如果错误地将一个可分配数组名进行了分配，则该数组以当前分配内存进行存储，而原数组中存储的内容被删除。

如果错误地将一个已经分配的指针数组进行了分配，则指针数组将按将分配内存进行存储，而原指针数组存储的内容被删除。

DEALLOCATE 语句又称释放语句，该语句的功能是把已分配给动态数组的内存空间释放掉，将该内存空间提供给系统另作它用。一个动态数组经过 DEALLOCATE 语句释放后，该动态数组分配的状态已不再存在，它立即变成了非当前分配状态。DEALLOCATE 语句的一般形式为：

DEALLOCATE(动态数组名 1, 动态数组名 2, …)

被释放的动态数组只须写一个数组名，一次可释放任意个数组。需要注意的是：如果被释放的数组是非当前分配状态时，则 DEALLOCATE 语句将造成程序的终止执行。因此，在使用 DEALLOCATE 语句时，必须谨慎考虑。下面是使用 DEALLOCATE 语句的例子：

DEALLOCATE(A,B)

当该语句被执行后，将释放掉动态数组 A,B 的内存，以后 A,B 不再按上述维界的数组使用，但 A,B 动态数组的性质不变，它可以再通过 ALLOCATE 语句分配新的维界投入使用。

## 5. 编程举例

**例 5.22** 设有矩阵  $A_{3 \times 4}, B_{4 \times 5}$ ，求  $A_{3 \times 4} \times B_{4 \times 5}$  之积矩阵  $C_{3 \times 5}$ 。

解：已知矩阵  $A \times B = C$  的操作是把  $A$  的第一行各元素与  $B$  第一列各元素成对相乘，而后把各对积之和作为  $C_{11}$ 。如果取  $A$  的第二行与  $B$  的第一列作上述操作，其和是  $C_{21}$ ，即：

$$C_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} + a_{14} \cdot b_{41} = \sum_{k=1}^4 a_{1k} b_{k1}$$

$$C_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} + a_{24} \cdot b_{41} = \sum_{k=1}^4 a_{2k} b_{k1}$$

一般地，

$$C_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + a_{i3} \cdot b_{3j} + a_{i4} \cdot b_{4j} = \sum_{k=1}^4 a_{ik} b_{kj}$$

也即对每个 C 元素下标  $i,j$ , 作  $k=1,4$  循环。为了求遍所有 C 元素, 又需对  $i$  作  $i=1,3$  循环, 对  $j$  作  $j=1,\dots,5$  循环, 形成一个三重循环。

为了使程序有普遍性, 编程时不采用常数组, 而是采用动态数组。因为前者只能求一种固定维界的矩阵的乘积, 而后者数组的维界可随时由读入的维界值决定而分配空间。假设要求  $A_{N \times L} \times B_{L \times M}$ , 则其积矩阵为  $C_{N \times M}$ , 只要保证 A 的第二维长与 B 的第一维长相等(均为 L), N, M 可以任意。程序如下:

```

PROGRAM MATMUL_A_B
IMPLICIT NONE
INTEGER :: N,M,L,I,J,K
REAL,DIMENSION (:,:) ALLOCATABLE :: A,B,C
READ *,N,J,M
ALLOCATE (A(1:N,1:L),B(L,M),C(N,M))
READ *,((A(I,J),J=1,L),I=1,N)
READ *,((B(I,J),J=1,M),I=1,L)
DO I=1,N
    DO J=1,M
        C(J,J)=0.0
        DO K=1,L
            C(I,J)=C(I,J)+A(I,K) * B(K,J)
        END DO
    END DO
    PRINT *,((C(I,J),J=1,M),I=1,N)
END PROGRAM MATMUL_A_B

```

程序先读入 M, L, N 等值, 随后以它们为维数的上界分配给动态数组 A, B, C, 使它们都有确定的维界, 然后读入 A, B(如无确定维界时无法读入), 读入时按行输入(第二个下标 J 在内循环)。求得数组 C 后按行打印方式输出, 以便阅读。

## 第九章 CM Fortran 数组变换

一个内部函数(或内部子例子程序),是程序语言预先定义的函数,CM FORTRAN 也不例外。CM FORTRAN 提供了丰富的内部函数库,数组操作丰富多彩,数组变换更是独具现代特色。它为解决复杂的科学计算和不同的应用问题,提供了一种功能强大的现代手段。CM FORTRAN 支持 FORTRAN 77 标准的全部内部函数和 FORTRAN 90 标准所增加的全部内部函数。对 CM FORTRAN 中出现的一切函数,既可作并行计算,也可用串行计算。若作串行计算,其效果与 FORTRAN 77 标准和 FORTRAN 90 标准大致相同。内部函数因其功能不同可分为三大类:基本函数、查询函数、变换函数。

**基本函数:**在基本函数中,它包含了数学上所有的初等函数,并且这些基本函数既可求简单函数值,也可求复杂的数组函数值。例如 A,B 都是形状相同的一维数组,若有  $B = \text{SQRT}(A)$ , 则它分别表示为  $B(1) = \text{SQRT}(A(1))$ ,  $B(2) = \text{SQRT}(A(2))\dots$ 。它可以推广到 N 维数组上。

**查询函数:**CM FORTRAN 的查询函数,有着多种多样的形式,如种类查询、字符查询、位查询、数组查询等等。

**变换函数:**可将已知的数组根据需要进行变换,得到各种形状不同的新数组。在 CM FORTRAN 中,数组变换按其功能又可分为四类:数据的移动,数组的归纳,数组的构造,数组的乘法等等。

CM FORTRAN 中的全部内部函数,均描述在 CM FORTRAN 文件库中。本章仅就最能反映那种结构的数据变换 在概要的入门初

组并行执行不同移位。

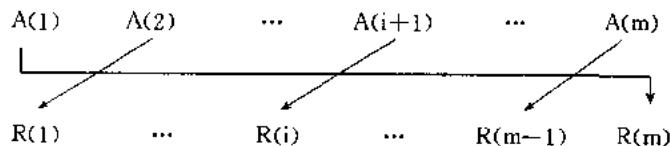
函数的功能：根据所给定的条件不同，对数组元素实现部分的并行移位或全部并行移位。

循环移位函数可分三种不同的情况：

(1) 所给循环移位函数形式为：

CSHIFT(ARRAY,DIM=1,SHIFT)

这时数组并行地执行一次排列。若限定 SHIFT 是一个非负的标量值时，我们假定数组的大小是 M，SHIFT 的值是 1，则数组元素循环移位规则如下所示：



在这里  $A(i)$  和  $R(i)$  分别表示数组的原元素和结果元素，各个箭头说明数据移动的情况，即这时所有数组元素向左循环移位。

**例 6.1** 设  $V = [1, 2, 3, 4, 5, 6, 7]$

则  $\text{CSHIFT}(V, \text{DIM}=1, \text{SHIFT}=1) \Rightarrow [2, 3, 4, 5, 6, 7, 1]$

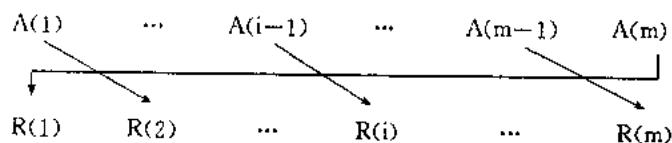
如果 SHIFT 的值是比 1 大，则循环移位的方向不变，只是数组元素多进行几次并行循环移位。

**例 6.2**  $\text{CSHIFT}(V, \text{DIM}=1, \text{SHIFT}=3) \Rightarrow [4, 5, 6, 7, 1, 2, 3]$

(2) 当 SHIFT 是一个负值时，循环移位函数的形式仍然为：

CSHIFT(ARRAY,DIM=1,SHIFT)

这时数组元素仍然是并行地执行一次排列，但所有的数组元素是向右循环移位。我们仍然取数组的大小为 M，SHIFT 的值是 -1，则数组元素的循环移位的规则如下图所示：



在这里箭头说明数据运动的情况。

**例 6.3**  $\text{CSHIFT}(V, \text{DIM}=1, \text{SHIFT}=-1) \Rightarrow [7, 1, 2, 3, 4, 5, 6]$

$\text{CSHIFT}(V, \text{DIM}=1, \text{SHIFT}=-2) \Rightarrow [6, 7, 1, 2, 3, 4, 5]$

(3) 当循环移位 CSHIFT 函数中的  $\text{DIM} > 1$  时，而且 SHIFT 是一个标量或是一个数组时，原数组移位的方式将各不相同。

**例 6.4** 设

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

则

$$\text{CSHIFT}(A, \text{DIM}=2, \text{SHIFT}=-1) \Rightarrow \begin{bmatrix} 5 & 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\text{CSHIFT}(A, \text{DIM}=2, \text{SHIFT}=[-1, 1, 0]) \Rightarrow \begin{bmatrix} 5 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

$$\text{CSHIFT}(V, \text{DIM}=2, \text{SHIFT}=[1, 2, 3]) \Rightarrow \begin{bmatrix} 2 & 3 & 4 & 5 & 1 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 5 & 1 & 2 & 3 \end{bmatrix}$$

当 SHIFT 取值为二维数组时, 则表示对原三维数组进行元素移位。三维数组的书写形式为:

$$C = \begin{bmatrix} 111 & 121 & 131 & 141 \\ 211 & 221 & 231 & 241 \\ 311 & 321 & 331 & 341 \\ 112 & 122 & 132 & 142 \\ 212 & 222 & 232 & 242 \\ 312 & 322 & 332 & 342 \end{bmatrix}$$

### § 6.1.2 截止移位 EOSHIFT 函数

截止移位函数的一般形式是:

`EOSHIFT(ARRAY, DIM, SHIFT[, BOUNDARY])`

其中: ARRAY 是一个数值类型数组或是一个逻辑数组。DIM 是一个整数类型的标量值, 它取值范围是:  $1 \leqslant \text{DIM} \leqslant n$ , 这里的 n 是数组的排列。SHIFT 是一个表达式, 当数组进行一次排列时, SHIFT 必须是一个标量, 其他情况下, SHIFT 可以是任意一个标量或者是数组; 如果 SHIFT 是一个数组, 则原数组进行重新移位。BOUNDARY 是一个可选项, 它表示与数组是相同的类型; 当数组是一次排列时, BOUNDARY 必须是标量, 其他情况它可以选择标量或是数组; BOUNDARY 的作用为, 在截止移位后数组元素空位由 BOUNDARY 中的元素填位。当 BOUNDARY 项省略时, 可用 BOUNDARY 的默认值填位, 默认值如下表:

原数组类型	BOUNDARY 的默认值
INTEGER	0
REAL	0.
DOUBLE PRECISION	0.d0
COMPLEX	(0., 0.)
DOUBLE COMPLEX	(0., 0.d0)
LOGICAL	.FALSE.

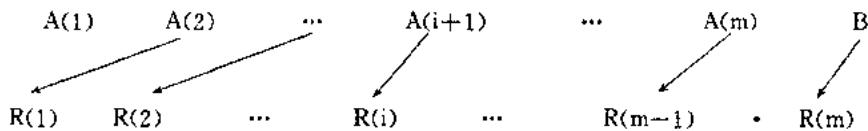
函数的功能: 根据所给的条件, 部分并行移位或全部并行移位。元素移出的部分由 BOUNDARY 中的元素来填位。截止移位的元素可以不同, 而且方向可以不同。

截止移位分三种不同的情况:

(1) 给定截止循环移位函数形式:

**EOSHIFT(ARRAY,DIM=1,SHIFT[,BOUNDARY])**

这时数组并行地执行一次排列。若限定 SHIFT 是一个非负的标量值, 我们假定 SHIFT 的值是 1, 数组的大小为 M, 则数组元素截止移动规则如下图所示:



这里  $A(i)$  和  $R(i)$  分别表示数组的原元素和结果元素,  $B$  为 BOUNDARY 中的元素。各个箭头标明数据移动的情况, 即这时所有数组元素向左移位。

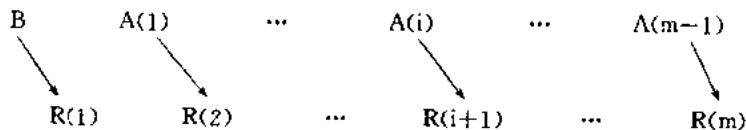
**例 6.5** 设  $V=[1,2,3,4,5,6]$

则:  $\text{EOSHIFT}(V, \text{DIM}=1, \text{SHIFT}=3) \Rightarrow [4,5,6,0,0,0]$

在这里, 一维数组  $V$  中的元素为整型, 所以 BOUNDARY 缺省时, 用 0 来填位。

又如:  $\text{EOSHIFT}(V, \text{DIM}=1, \text{SHIFT}=2, \text{BOUNDARY}=99) \Rightarrow [3,4,5,6,99,99]$

(2) 当 SHIFT 是一个负值时, 截止移位的形式仍为:  $\text{EOSHIFT}(\text{ARRAY}, \text{DIM}=1, \text{SHIFT})$ , 这时数组执行一次排列, 但所有的数组元素是向右截止移位。我们仍然假定数组的大小是  $m$ ,  $\text{SHIFT}=-1$ , 则数组元素截止移位规则如下图所示:



在这里箭头表示数据运动的情况,  $B$  为 BOUNDARY 的元素。

**例 6.6**  $\text{EOSHIFT}(V, \text{DIM}=1, \text{SHIFT}=-3, \text{BOUNDARY}=99) \Rightarrow [99,99,99,1,2,3]$

$$\text{又如: 设 } A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$$

$$\text{则 } \text{EOSHIFT}(A, \text{DIM}=1, \text{SHIFT}=-1, \text{BOUNDARY}=[1:4]) \Rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \end{bmatrix}$$

在这里, 数组是二维时,  $\text{DIM}=1, \text{SHIFT}$  是负值时表示向下截止移位。

(3) 当截止移位 EOSHIFT 函数中的  $\text{DIM}>1$ , 而且 SHIFT 是一个标量或是一个数组时, 原数组移位的方式将各不相同。

$$\text{例 6.7} \text{ 设 } A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$$

$$\text{则 } \text{EOSHIFT}(A, \text{DIM}=2, \text{SHIFT}=-1, \text{BOUNDARY}=[1:3]) \Rightarrow \begin{bmatrix} 1 & 11 & 12 & 13 \\ 2 & 21 & 22 & 23 \\ 3 & 31 & 32 & 33 \end{bmatrix}$$

$$\text{EOSHIFT}(A, \text{DIM}=1, \text{SHIFT}=[-2, -1, 0, 1]) \Rightarrow \begin{bmatrix} 0 & 0 & 13 & 24 \\ 0 & 12 & 23 & 34 \\ 11 & 22 & 33 & 0 \end{bmatrix}$$

在这里, SHIFT 中各元素分别对应数组 A 中各列所截止移位的情况。即: SHIFT 中的第一个数 -2, 表示对数组 A 中第一列的元素向下截止移位; SHIFT 中的第二个数 -1, 表示对数组 A 中第二列的元素向下截止移位; SHIFT 中的第三个数 0, 表示对数组 A 中第三列的元素不移位; SHIFT 中第四个数 1, 表示对数组 A 中第四列的元素向上截止移位。

$$\text{又如 } \text{EOSHIFT}(A, \text{DIM}=2, \text{SHIFT}=[-1, 0, +1]) \Rightarrow \begin{bmatrix} 0 & 11 & 12 & 13 \\ 21 & 22 & 23 & 24 \\ 32 & 33 & 34 & 0 \end{bmatrix}$$

在这里, SHIFT 中各元素分别对应数组 A 中各行所截止移行的情况。

$$\text{例 6.8 设 } A = \begin{bmatrix} 111 & 121 & 131 & 141 \\ 211 & 221 & 231 & 241 \\ 311 & 321 & 331 & 341 \\ 112 & 122 & 132 & 142 \\ 212 & 222 & 232 & 242 \\ 321 & 322 & 332 & 342 \end{bmatrix} \quad S = \begin{bmatrix} -1 & 1 \\ 0 & 2 \\ 1 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

$$\text{则: } \text{EOSHIFT}(A, \text{DIM}=2, \text{SHIFT}=S, \text{BOUNDARY}=B) \Rightarrow \begin{bmatrix} 1 & 111 & 121 & 131 \\ 211 & 221 & 231 & 241 \\ 321 & 331 & 341 & 3 \\ 122 & 132 & 142 & 4 \\ 232 & 242 & 5 & 5 \\ 342 & 6 & 6 & 6 \end{bmatrix}$$

在截止移位的过程中, S 中的第一列各元素, 表示对三维数组表中上半部各行元素的移位情况, 正值向左截止移位, 负值向右截止移位; B 中第一列各元素表示对三维数组表中上半部各行的填充数。而 S 中的第二列各元素, 表示对三维数组表中下半部各行元素截止移行情况; B 中第二列各元素表示对三维数组表上下半部各行的填充数。

### § 6.1.3 矩阵的转置函数

矩阵的转置函数的一般形式是:

TRANSPOSE(MATRIX)

其中: MATRIX 是需要转置的二维矩阵元素。元素可以是数值或者是逻辑类型。

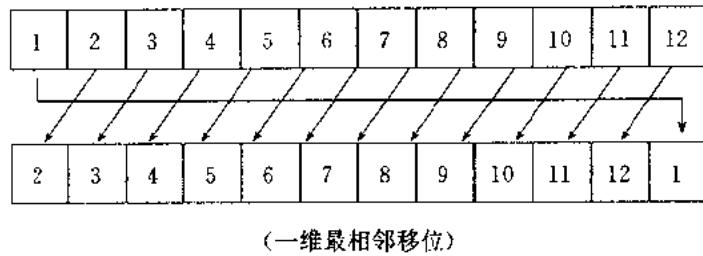
函数的功能: 通过转置后, 将原二维矩阵中的元素行变为列, 列变为行。

$$\text{例 6.9 } \text{TRANSPOSE} \left( \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \right) \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

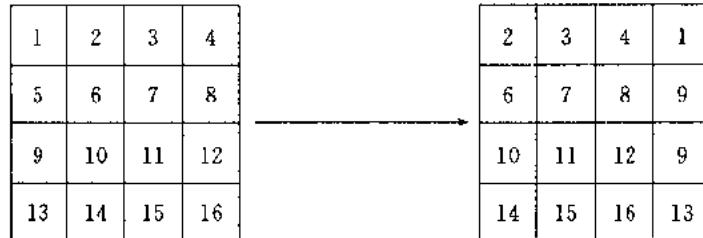
$$\text{TRANSPOSE} \left( \begin{bmatrix} T & \cdot & T \\ \cdot & T & \cdot \end{bmatrix} \right) \Rightarrow \begin{bmatrix} T & \cdot \\ \cdot & T \\ T & \cdot \end{bmatrix}$$

在这里, 逻辑类型矩阵的元素“T”表示真值, “·”表示假值。

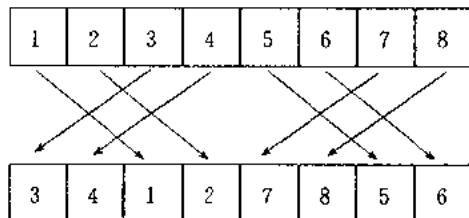
综上所述, 我们将数据移动简略归纳如下图所示:



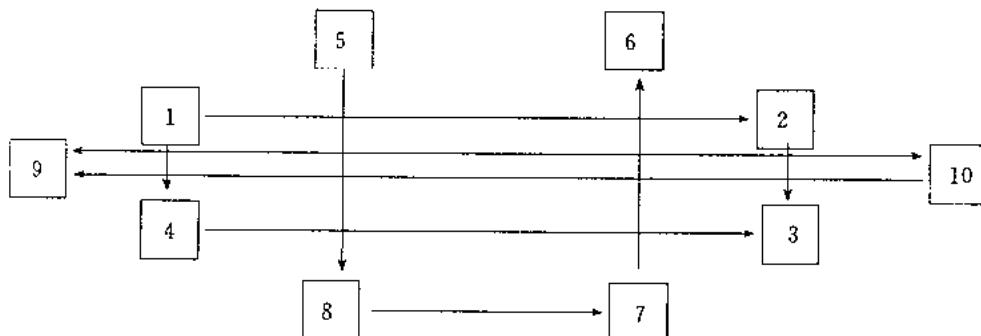
(一维最相邻移位)



(二维行/列移位)



(蝶式移位)



(完全不规则移位)

## § 6.2 数组的归约函数

数组归约函数是对数值数组和逻辑数组进行运算的操作函数。它适应于那些只需对数组部分元素进行运算操作,而其他部分元素将被屏蔽,减少数组元素的运算法量,提高程序的执行速度。数组归约函数包括:数值归约函数 MAXVAL,MINVAL,SUM,PRODUCT 和逻辑归约函数 ANY,ALL,COUNT。在这一节里,我们分别介绍归约函数的一般形式和运算操作举例。

### § 6.2.1 求数组中最大元素的值函数

MAXVAL 函数是并行地计算一个数值数组中的全部元素里的最大值,或是并行地求部分数组中的元素里的最大值。MAXVAL 函数的一般形式为:

MAXVAL(ARRAY[,DIM][,MASK])

其中: ARRAY 是一个整数类型或实数类型的数组; DIM 是一个可选项,它是一个整数类型的常量表达式。DIM 的取值范围: $1 \leqslant \text{DIM} \leqslant n$ , 在这里  $n$  是数组的排列。可选项 MASK 是一个逻辑数组或者是一个逻辑条件表达式,它若是逻辑数组,则必须与 ARRAY 项相一致,该 MASK 项起屏蔽作用的。当 ARRAY 是逻辑数组时,它的元素只包含两种:“真”与“假”,“真”元素用“T”表示,“假”元素用“.”表示。此处的作用是只保留数组 ARRAY 中对应于 MASK 中为“真”的元素参与运算,其他元素被屏蔽掉。若 MASK 是一个逻辑条件表达式,作用是:凡满足条件的元素,就参与运算,不满足条件的元素则被屏蔽掉,不参加运算。MASK 项的使用有普遍意义,很多数组的内部函数都有可选(或必选)项 MASK,它的作用都如上述,以后不再重叙 MASK 的作用。

MAXVAL 函数的功能:求出所给的数组中一个或几个最大元素值。

MAXVAL 函数分三种不同的形式:

(1) 当数组是一维时,并且 DIM 是预置的,它的形式为:

MAXVAL(ARRAY,DIM=1[,MASK])

此时所求函数值是一维数组中最大者。

例 6.10  $K = \text{MAXVAL}([2, 3, -5, 8, 4, 9], \text{DIM}=1) \Rightarrow K = 9$

若  $A = [2, 3, -5, 7], M = [\cdot, T, T, \cdot]$

则  $K = \text{MAXVAL}(\text{ARRAY}=A, \text{DIM}=1, \text{MASK}=M) \Rightarrow K = 3$

(2) 当数组是二维的,并且 DIM 缺省时,它的一般形式为:

MAXVAL(ARRAY[,MASK])

此时所求的函数值为二维数组全部或部分元素的最大者。

例 6.11 设  $A = \begin{bmatrix} 1 & 5 & 3 & 7 \\ 4 & 2 & 6 & 3 \\ 9 & 2 & 1 & 5 \end{bmatrix}$

则  $I = \text{MAXVAL}(A) \Rightarrow I = 9$

$J = \text{MAXVAL}(A(:, 2:3)) \Rightarrow J = 6$

在这里,  $J$  的值是从数组 A 中的第二至第三列中选取的最大元素。

又如:  $L = \text{MAXVAL}\left(\begin{bmatrix} -4 & 25 & -1 \\ 15 & 5 & -2 \end{bmatrix}\right)$ ,  $\text{MASK} = \begin{bmatrix} T & \cdot & T \\ \cdot & T & T \end{bmatrix} \Rightarrow L = 5$

此时  $L$  的值是从 MASK 中为真的数组中选取的最大元素。

(3) 当数组是二维时,并且 DIM 是预置的,MAXVAL 函数求得最大元素的值是一维数组。

例 6.12 若  $B = \begin{bmatrix} 3 & 1 & 5 & 18 \\ 17 & -7 & 14 & 0 \end{bmatrix}$

则  $\text{MAXVAL}(B, \text{DIM}=1) \Rightarrow [17, 1, 14, 18]$

这里所求得的最大元素值分别是各列中的最大者。

$\text{MAXVAL}(B, \text{DIM}=2) \Rightarrow [18, 17]$

此时所求的函数组是数值 B 中各行中的最大者。

$\text{MAXVAL}(B(:, 2:3), \text{DIM}=2) \Rightarrow [5, 14]$

此时所求的函数值是数组 B 中从第一行中的第二列至第三列中选取最大元素 5; 从数组 B 中第二行中的第二列至第三列中选取最大元素 14。

$$\text{若 } M = \begin{bmatrix} \cdot & \cdot & \cdot & T \\ T & \cdot & T & \cdot \end{bmatrix}, \quad N = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ T & T & T & T \end{bmatrix}$$

则  $\text{MAXVAL}(B, \text{DIM}=2, \text{MASK}=M) \Rightarrow [18, 17]$

$\text{MAXVAL}(B, \text{DIM}=1, \text{MASK}=N) \Rightarrow [17, -7, 14, 0]$

$\text{MAXVAL}(B, \text{DIM}=1, \text{MASK}=M) \Rightarrow [17, -2147483648, 14, 18]$

此时所求的函数值中,  $-2147483648$  是数组 B 第二列元素全部被屏蔽掉, 计算机自动产生一个最大负数来填位。

注: 求数组中最小元素值函数  $\text{MINVAL}$  的规则与求  $\text{MAXVAL}$  规则相同, 在这里不赘述。

### § 6.2.2 数组的乘积

数组的乘积函数是并行地计算数值数组中全部元素或部分元素的乘积之和。数组乘积函数的一般形式为:

$\text{PRODUCT}(\text{ARRAY}[,\text{DIM}][,\text{MASK}])$

其中:  $\text{ARRAY}$  是一个数值类型的数组; 可选项  $\text{DIM}$  是一个整数类型的常量表达式,  $\text{DIM}$  的取值范围是:  $1 \leq \text{DIM} \leq n$ , 在这里  $n$  是数组的排列; 可选项  $\text{MASK}$  是一个逻辑类型的数组或是一个逻辑类型标量表达式, 如果  $\text{MASK}$  是一数组, 则必须与所求数组一致。

$\text{PRODUCT}$  函数功能: 求所给数组的部分或全部的乘积(注意:  $\text{PRODUCT}$  函数乘积与矩阵乘法不同, 请注意区别)。

$\text{PRODUCT}$  函数分三种情况:

(1)  $\text{DIM}$  缺省时的一般形式:

$\text{PRODUCT}(\text{ARRAY}[,\text{MASK}])$

这时函数的值为  $\text{MASK}$  项中为“真”的对应元素乘积之和。

例 6.13  $\text{PRODUCT}([2, 3, -5, 7]) \Rightarrow -210$

$\text{PRODUCT}([2, 3, -5, 7], \text{MASK}=[T, T, \cdot, T]) \Rightarrow 42$

$\text{PRODUCT} \begin{bmatrix} -4 & 2 & -1 \\ 3 & 5 & -2 \end{bmatrix}, \text{MASK} = \begin{bmatrix} T & T & \cdot \\ \cdot & T & T \end{bmatrix} \Rightarrow 80$

(2) 当数组是一维且  $\text{DIM}=1$  时, 所求元素是  $\text{MASK}$  中为“真”值对应元素乘积之和。

例 6.14  $\text{PRODUCT}([2, 3, -5, 7], \text{DIM}=1) \Rightarrow -210$

$\text{PRODUCT}([2, 3, -5, 7], \text{DIM}=1, \text{MASK}=[T, \cdot, \cdot, T]) \Rightarrow 14$

(3) 当数组是二维且  $\text{DIM}$  是预置的时, 所求函数值为对应数组的行(或列)乘积之和(即函数值为一组数)。

例 6.15 若  $B = \begin{bmatrix} 3 & 2 & 5 \\ 4 & 1 & -7 \end{bmatrix}$

$$\text{则 } \text{PRODUCT}(B, \text{DIM}=2, \text{MASK} = \begin{bmatrix} T & T & \cdot \\ \cdot & T & T \end{bmatrix}) \Rightarrow [6, -7]$$

$$\text{PRODUCT}(B, \text{DIM}=1, \text{MASK} = \begin{bmatrix} T & T & \cdot \\ \cdot & T & T \end{bmatrix}) \Rightarrow [3, 2, -7]$$

$$\text{PRODUCT}(B, \text{DIM}=2, \text{MASK} = \begin{bmatrix} \cdot & \cdot & \cdot \\ T & T & T \end{bmatrix}) \Rightarrow [4, 1, -7]$$

### § 6.2.3 求和函数

数组的求和函数是并行地计算数值数组中全部元素或部分元素相加之和。数组的求和函数的一般形式为：

SUM(ARRAY[, DIM][, MASK])

其中：ARRAY 是一个数值类型的数组；可选项 DIM 是一个整数类型的常量表达式，它的取值范围是： $1 \leqslant \text{DIM} \leqslant n$ ，在这里  $n$  是数组的一种排列；可选项 MASK 是一个逻辑数组或是一个标量表达式，若是一个数组，则必须与所求数组一致。

SUM 函数功能：并行地计算数组全部元素或部分元素之和。

求和函数分两种情况：

(1) 当数组是一维数组并且  $\text{DIM}=1$ ，或数组多维且  $\text{DIM}$  缺省，这种情况下所求函数值是数组全部元素相加之和。

**例 6.16**  $\text{SUM}([2, 3, -5, 7], \text{DIM}=1) \Rightarrow 7$

$\text{SUM}([2, 3, -5, 7], \text{DIM}=2, \text{MASK} = [T, \cdot, \cdot, T]) \Rightarrow 9$

若设  $A = \begin{bmatrix} -4 & 2 & -1 \\ 3 & 5 & -2 \end{bmatrix}$

则  $\text{SUM}(A) \Rightarrow 3$

$\text{SUM}(A, \text{MASK} = A > 0) \Rightarrow 10$

又如：若  $B = \begin{bmatrix} 1 & 4 & -7 & 10 \\ 2 & -5 & 8 & -11 \\ 3 & 6 & 9 & -12 \end{bmatrix}$ ,  $M = \begin{bmatrix} T & T & \cdot & \cdot \\ \cdot & \cdot & T & T \\ \cdot & T & T & \cdot \end{bmatrix}$

则  $\text{SUM}(B, \text{MASK} = B > 0) \Rightarrow 43$

$\text{SUM}(B, \text{MASK} = M) \Rightarrow 17$

此处的数组和是 MASK 中为“真”值对应数组元素之和。

(2) 当数组维数大于 1 时，DIM 是预置的，这时所求数组值为一组数。

**例 6.17** 若  $A = \begin{bmatrix} 3.5 & 2.0 & 5.0 \\ 4.5 & 1.5 & -7.0 \end{bmatrix}$

则  $\text{SUM}(A, \text{DIM}=1) \Rightarrow [8.0, 3.5, -2.0]$

这里是按列求组数和。

$\text{SUM}(A, \text{DIM}=2) \Rightarrow [10.5, -1.0]$

此时数组和是按行求和的。

$$\text{SUM}(A, \text{DIM}=2, \text{MASK} = \begin{bmatrix} T & T & \cdot \\ \cdot & T & T \end{bmatrix}) \Rightarrow [5.5, -5.5]$$

$$\text{SUM}(A, \text{DIM}=1, \text{MASK} = \begin{bmatrix} \cdot & \cdot & \cdot \\ T & T & T \end{bmatrix}) \Rightarrow [4.5, 1.5, -7.0]$$

### § 6.2.4 计数函数

计数函数是并行地统计一个逻辑数组中为 ·TRUE· 元素的个数的多少。计数函数的一般形式为：

COUNT(MASK[,DIM])

其中：MASK 是一个逻辑类型的数组；可选项 DIM 是一个整数类型的标量表达式，它的取值范围是： $1 \leqslant \text{DIM} \leqslant N$ ，在这里 N 是逻辑数组 MASK 的一种排列。

COUNT 函数的功能：统计逻辑数组中“真”值数。

例 6.18 COUNT([T, ·, T]) $\Rightarrow 2$

COUNT([T, ·, T], DIM=1) $\Rightarrow 2$

COUNT(MASK=[T, ·, T, T], DIM=1) $\Rightarrow 3$

COUNT([ · T · ]  
[ T · T ]) $\Rightarrow 3$

若  $B = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ ,  $C = \begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$

则 COUNT(B . EQ. C, DIM=1) $\Rightarrow [0, 2, 1]$

在这里 COUNT 函数是统计数组 B 和数组 C 对应的各列相等的元素的个数。

COUNT(B . EQ. C, DIM=2) $\Rightarrow [2, 1]$

该函数统计两个数组对应行中的相等元素的个数。

### § 6.2.5 ALL 和 ANY 函数

像其他逻辑类型的归约函数一样，ALL 函数和 ANY 函数是按照指定的大小来对一个逻辑数组进行全部或部分进行归约的，返回的结果是一个逻辑值或一组逻辑值。ALL 函数与 ANY 函数功能略有不同，下面分别进行介绍。

(1) ALL 函数的一般形式为

ALL(MASK[,DIM])

其中：MASK 是一个逻辑数组；可选项 DIM 是一个整型标量表达式，当  $\text{DIM}=1$  时，对逻辑数组按每列进行归约，只有当全列逻辑元素为 TRUE 时，返回结果才为 TRUE，否则，返回结果为 FALSE。当  $\text{DIM}=2$  时，对逻辑数组按行进行归约，只有当全行逻辑元素为 TRUE 时，返回结果才为 TRUE，否则，返回结果为 FALSE。当 DIM 缺省，只有逻辑数组中所有元素为 TRUE 时，返回结果才为 TRUE。否则，返回结果为 FALSE。

例 6.19 ALL([T, ·, T]) $\Rightarrow F$

ALL([T, T, T]) $\Rightarrow T$

ALL([ T T T  
· T · ], DIM=1) $\Rightarrow [ ·, T, · ]$

ALL([ T T T  
· T · ], DIM=2) $\Rightarrow [ T, · ]$

若  $B = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ ,  $C = \begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$

则 ALL(B . EQ. C, DIM=1) $\Rightarrow [ ·, T, · ]$

$\text{ALL}(B, \text{EQ}, C, \text{DIM}=2) \Rightarrow [\cdot, \cdot]$

(2) ANY 函数的一般形式为:

$\text{ANY}(\text{MASK}, \text{DIM})$

其中: MASK 是一个逻辑数组; 可选项 DIM 是一个整型标量表达式, 当  $\text{DIM}=1$  时, 是对逻辑数组按列进行归约。只要归约的列中有一个元素为 TRUE, 则返回的结果就是 TRUE。只有归约的列中全部元素为 FALSE, 返回结果才为 FALSE。当  $\text{DIM}=2$  时, 是对逻辑数组按行进行归约。只有全行元素为 FALSE 时, 返回结果才为 FALSE; 否则, 返回结果为 TRUE。当 DIM 缺省时, 只有全部数组元素为 FALSE, 返回的结果才为 FALSE; 否则, 返回结果为 TRUE。

例 6.20  $\text{ANY}([\cdot, \cdot, T]) \Rightarrow T$

$\text{ANY} \left[ \begin{array}{ccc} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{array} \right] \Rightarrow F$

若  $B = \begin{bmatrix} \cdot & T & \cdot & T \\ \cdot & T & \cdot & T \\ \cdot & T & \cdot & T \end{bmatrix}$

则  $\text{ANY} = (\text{MASK} = B) \Rightarrow T$

$\text{ANY} = (B, \text{DIM}=1) \Rightarrow [\cdot, T, \cdot, T]$

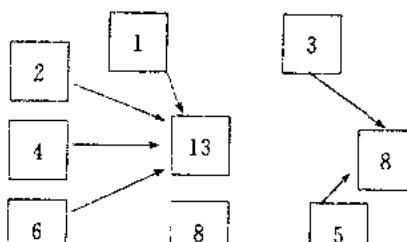
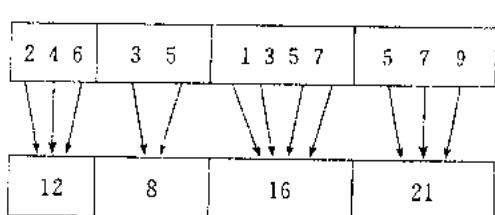
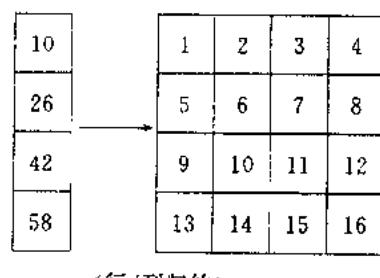
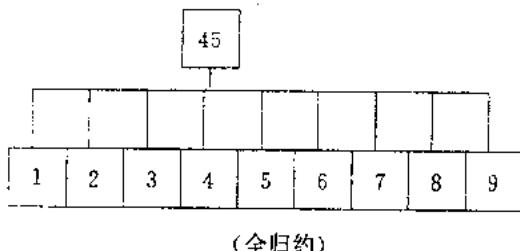
$\text{ANY} = (B, \text{DIM}=2) \Rightarrow [T, T, T]$

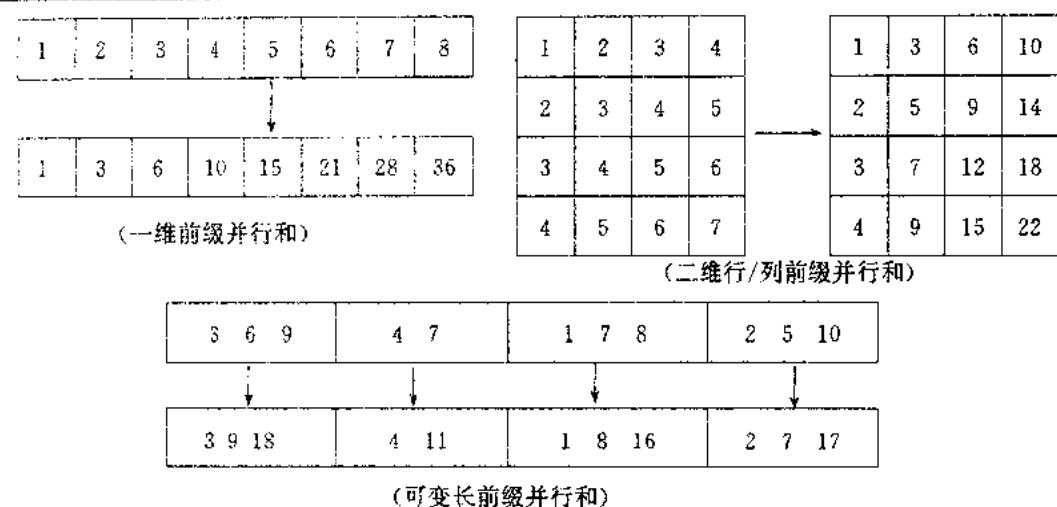
又如设  $B = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}, C = \begin{bmatrix} 0 & 4 & 5 \\ 7 & 3 & 8 \end{bmatrix}$

则  $\text{ANY}(B, \text{EQ}, C, \text{DIM}=1) \Rightarrow [\cdot, \cdot, T]$

$\text{ANY}(B, \text{EQ}, C, \text{DIM}=2) \Rightarrow [T, \cdot]$

综上所述, 我们将数据归约概要归纳如下图:





### § 6.3 数组的构造函数

数组的构造函数是按照现有数组元素根据所给的条件,构造一个新的 CM 数组来。它包括如下函数:

- 对角线构造数组 DIAGONAL 函数: 它将根据给定的一向量值作为构造一个矩阵结构中的对角线元素, 矩阵的元素由规定值填满(或缺省)值。
- 合并数组 MERGE 函数: 它将两个一致的数组合并成一个新的数组。
- 数组的压缩 PACK 函数和数组的扩散 UNPACK 函数: PACK 函数将大小为 N 维数组压缩成一向量, UNPACK 函数将一向量分散成 N 维数组。
- 复制 REPLICATE 函数和数组扩展 SPREAD 函数: REPLICATE 函数是根据给定的数组重新复制生成一个新数组。SPREAD 函数是根据所给一个数值或数组扩展成一个新数组。
- 重新整形数组 RESHAPE 函数, 它按照给定数组构造一个形状不同的新数组。

在这一节里, 介绍上述各数组构造函数。

#### § 6.3.1 对角线构造数组函数

对角线构造数组 DIAGONAL 函数的一般形式为:

DIAGONAL(ARRAY[, FILL])

其中: ARRAY 是一个数值类型或者是逻辑类型的一维数组; 可选项 FILL 是与数组类型相同的标量, 当它缺省时, 默认如下的值:

数组的类型	填充的默认值
INTEGER	0
REAL	0.
DOUBLE PRECISION	0.d0
COMPLEX	(0,0)
DOUBLE COMPLEX	(0.,0d0)
LOGICAL	.FALES.

例 6.20 选项 FILL 作用是填满矩阵对角线以外的位置。

对角线构造数组函数功能：根据给定的一维数组，构造一个对角线矩阵。

例 6.21 当可选项 FILL 缺省时，生成对角线矩阵：

$$\text{DIAGONAL}([1, 2, 3]) \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

$$\text{DIAGONAL}([11., 22., 33.]) \Rightarrow \begin{bmatrix} 11. & 0. & 0. \\ 0. & 22. & 0. \\ 0. & 0. & 33. \end{bmatrix}$$

$$\text{DIAGONAL}([3[T]]) \Rightarrow \begin{bmatrix} T & \cdot & \cdot \\ \cdot & T & \cdot \\ \cdot & \cdot & T \end{bmatrix}$$

例 6.22 当可选项预置值时，生成对角线矩阵

$$\text{DIAGONAL}([12.3, 13.4, 14.5], [\text{FILL} = -1.]) \Rightarrow \begin{bmatrix} 12.3 & -1. & -1. \\ -1. & 13.4 & -1. \\ -1. & -1. & 14.5 \end{bmatrix}$$

$$\text{DIAGONAL}([77, 88, 99], \text{FILL} = 2) \Rightarrow \begin{bmatrix} 77 & 2 & 2 \\ 2 & 88 & 2 \\ 2 & 2 & 99 \end{bmatrix}$$

### 6.3.2 数组归并构造函数

数组归并构造函数的一般形式为：

MERGE(TSOURCE, FSOURCE, MASK)

其中，TSOURCE, FSOURCE 具有相同类型数组或常量表达式，它可以是一个数值类型或者是一个逻辑类型，MASK 是一个逻辑类型表达式。

该函数功能：MASK 中为“真”值的项就选 TSOURCE 中元素为归并数组中对应的元素。（注意：这里不是两个数组对应元素相加。）

$$\text{例 6.23 } \text{MERGE}\left(\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}, \begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}, \begin{bmatrix} T & \cdot & T \\ \cdot & \cdot & T \end{bmatrix}\right) \Rightarrow \begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$$

在这里 MASK 中有三项为“真”值的项，所以对应选取  $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}$  中第一行第一列中 1，第一行第三列中 5，第二行第三列中 6，放到归并数组对应位置上，其他由  $\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}$  对应项来填位。

例 6.24  $\text{MERGE}(0, 1, \text{MOD}([1:7], 2) = 1) \Rightarrow [0, 1, 0, 1, 0, 1, 0]$ ，在这里  $\text{MASK} = [T, \cdot, T, \cdot, T, \cdot, T]$ ，当 MASK 中为“真”值的项取第一个元素，为“假”值的项取第二个元素 1，所以构成新的数组为  $[0, 1, 0, 1, 0, 1, 0]$ 。

### § 6.3.3 数组的压缩与扩散函数

(1) 压缩函数的一般形式是：

PACK(ARRAY, MASK[, VECTOR])

其中: ARRAY 是一个数值类型的数组或是一个逻辑类型的数组; MASK 是一个逻辑类型的标量或是与 ARRAY 相一致的数组; 可选项 VECTOR 是与 ARRAY 同类的一维数组。VECTOR 中的个数,最少不得少于屏蔽项 MASK 中为 TRUE 的元素个数。若 MASK 是一个标量值 TRUE,则 VECTOR 中必须最少可以存放下数组中的全部元素。VECTOR 作用是事先为数组压缩后存放元素留下的存放空间。

压缩函数的功能:在 MASK 的控制下,从 ARRAY 中选取符合条件的元素进行集中,存放到一维数组中。

若数组压缩后元素的个数少于 VECTOR 中所设置的单元,这时用 VECTOR 中的元素来填充。

PACK 函数执行前后结果可用如下图形说明:

MASK	•	•	T	•	•	T	T	...	T	...	T	•		
ARRAY	•	•	a <sub>1</sub>	•	•	a <sub>2</sub>	a <sub>3</sub>	...	a <sub>1</sub>	...	a <sub>t</sub>	•		
RESULT	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	...	a <sub>t</sub>	...	a <sub>t</sub>	V <sub>t+1</sub>	V <sub>t+2</sub>	...	V <sub>k</sub>	...	V <sub>n-1</sub>	V <sub>n</sub>
VECTOR	V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	...	...	...	V <sub>t</sub>	V <sub>t+1</sub>	V <sub>t+2</sub>	...	V <sub>k</sub>	...	V <sub>n-1</sub>	V <sub>n</sub>

其中:RESULT 表示数组压缩后结果。

例 6.25 设  $A = \begin{bmatrix} 0 & -1 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$  由压缩函数将非零元素集中起来,

则:  $\text{PACK}(A, \text{MASK}=A . \text{NE. } 0) \Rightarrow [9, -1, 7]$  (注意:因为 FORTRAN 存放数组元素是以列存放的顺序,所以在压缩的过程中,将一列一列的非零元素顺序地存放到一维数组中。)

又如:  $\text{PACK}(A, A . \text{NE. } 0, [1:10]) \Rightarrow [9, -1, 7, 4, 5, 6, 7, 8, 9, 10]$ , 在这里,由于有选项 VECTOR,并且 VECTOR=[1,2,3,4,5,6,7,8,9,10]是一维数组,而数组 A 中非零元素只有三个,通过压缩,只顺序地占去三个位置,剩下位置元素不动。

$$\text{若 } B = \begin{bmatrix} 7 & 14 & -9 \\ -5 & 23 & 56 \end{bmatrix}$$

则  $\text{PACK}(B, B > 0) \Rightarrow [7, 14, 23, 56]$

$\text{PACK}(B, B < 0, [5[0]]) \Rightarrow [-5, -9, 0, 0, 0]$

在这里,  $[5[0]] = [0, 0, 0, 0, 0]$

## (2) 数组的扩散函数

数组的扩散函数 UNPACK 与压缩函数 PACK 的目的相反,它把一维数组中的元素按照一定的要求分散成任意维数组。扩散函数 UNPACK 的一般形式为:

UNPACK(VECTOR, MASK, FIELD)

其中:VECTOR 是一个数值类型的一维数组或者是一个逻辑类型的一维数组。我们假定 VECTOR 中的元素的个数为 T,而这个 T 必须与逻辑数组 MASK 中为“真”的元素个数相同。MASK 是一个逻辑类型的数组表示式,作用是:规定扩散成维数的大小,并将一维数组 VECTOR 中元素分布在逻辑数组 MASK 里为“T”值的分布情况。FIELD 是与 VECTOR

类型相同的一个标量或数组。若是数组,那么它的元素个数要与 MASK 中元素个数一致。FIELD 的作用是,将 MASK 中为“假”的值的位置填数字。

UNPACK 函数的功能: 将一维数组 VECTOR 中的元素,按照逻辑数组 MASK 中的要求,分散到 MASK 中为“T”值的位置上,剩下的位置由 FIELD 的元素填满。UNPACK 的作用以图示说明如下:

VECTOR	<table border="1"> <tr><td>V<sub>1</sub></td><td>V<sub>2</sub></td><td>...</td><td>V<sub>i</sub></td><td>...</td><td>V<sub>t</sub></td><td>...</td></tr> </table>	V <sub>1</sub>	V <sub>2</sub>	...	V <sub>i</sub>	...	V <sub>t</sub>	...				
V <sub>1</sub>	V <sub>2</sub>	...	V <sub>i</sub>	...	V <sub>t</sub>	...						
RESULT	<table border="1"> <tr><td>f<sub>1</sub></td><td>f<sub>2</sub></td><td>V<sub>1</sub></td><td>f<sub>4</sub></td><td>f<sub>5</sub></td><td>V<sub>2</sub></td><td>...</td><td>V<sub>i</sub></td><td>...</td><td>V<sub>t</sub></td><td>...</td></tr> </table>	f <sub>1</sub>	f <sub>2</sub>	V <sub>1</sub>	f <sub>4</sub>	f <sub>5</sub>	V <sub>2</sub>	...	V <sub>i</sub>	...	V <sub>t</sub>	...
f <sub>1</sub>	f <sub>2</sub>	V <sub>1</sub>	f <sub>4</sub>	f <sub>5</sub>	V <sub>2</sub>	...	V <sub>i</sub>	...	V <sub>t</sub>	...		
FIELD	<table border="1"> <tr><td>f<sub>1</sub></td><td>f<sub>2</sub></td><td>f<sub>3</sub></td><td>f<sub>4</sub></td><td>f<sub>5</sub></td><td>f<sub>6</sub></td><td>...</td><td>f<sub>k</sub></td><td>...</td><td>f<sub>t</sub></td><td>...</td></tr> </table>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>	...	f <sub>k</sub>	...	f <sub>t</sub>	...
f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>	...	f <sub>k</sub>	...	f <sub>t</sub>	...		
MASK	<table border="1"> <tr><td>•</td><td>•</td><td>T</td><td>•</td><td>•</td><td>T</td><td>...</td><td>T</td><td>.....</td></tr> </table>	•	•	T	•	•	T	...	T	.....		
•	•	T	•	•	T	...	T	.....				

扩散函数 UNPACK 分两种情况:

①当 FIELD 是一个标量时,扩散函数将一维数组 VECTOR 中元素按 MASK 的要求分散到多维数组上,剩下的所有位置都由 FIELD 中的同一标量来填满。

$$\text{例 6.26 } \text{UNPACK}([11, 22, 33]), \text{ MASK} = \begin{bmatrix} \cdot & T \\ T & \cdot \\ \cdot & T \end{bmatrix}, \text{ FIELD} = 7 \Rightarrow \begin{bmatrix} 7 & 22 \\ 11 & 7 \\ 7 & 33 \end{bmatrix}$$

此处的分散过程是以一列一列的来扩散。

$$\text{又如: } \text{UNPACK}([11, 22, 33, 44, 55]), \text{ MASK} = \begin{bmatrix} \cdot & \cdot & T \\ \cdot & T & \cdot \\ T & T & T \end{bmatrix}, \text{ FIELD} = 9 \Rightarrow \begin{bmatrix} 9 & 9 & 44 \\ 9 & 22 & 9 \\ 11 & 33 & 55 \end{bmatrix}$$

②当 FIELD 是一个与 VECTOR 类型相同的数组时,FIELD 数组中的元素个数要与 MASK 中元素个数要一致。

$$\text{例 6.27 } \text{UNPACK}([11, 22, 33]), \text{ MASK} = \begin{bmatrix} \cdot & T \\ T & \cdot \\ \cdot & T \end{bmatrix}, \text{ FIELD} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 22 \\ 11 & 5 \\ 3 & 33 \end{bmatrix}$$

#### § 6.3.4 复制函数和扩展函数

##### (1) 复制函数 REPLICATE

将给定的数组,按照不同的要求重复进行拷贝复制。该函数的一般形式为:

REPLICATE(ARRAY, DIM, NCOPIES)

其中,ARRAY 是一个数值类型的数组,或是一个逻辑类型的数组;DIM 是一个整数类型的标量表达式,它的取值范围在:  $1 \leqslant \text{DIM} \leqslant N$ , 在这里的 N 是对数组 ARRAY 的一种排列数;NCOPIES 是一个整型标量表达式,它的作用是重复复制的次数。

该函数的功能:对所给定的数组进行复制。

例 6.28 将给定的数组要求按照行的方向进行复制:

$\text{REPLICATE}(\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}, \text{DIM}=2, \text{NCOPIES}=3) \Rightarrow \begin{bmatrix} 1 & 3 & 1 & 3 & 1 & 3 \\ 2 & 4 & 2 & 4 & 2 & 4 \end{bmatrix}$

在此处, 数组  $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$  被复制三次。

当给定的数组要求按照列的方向复制:

$\text{REPLICATE}(\begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix}, \text{DIM}=1, \text{NCOPIES}=2) \Rightarrow \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$

若要求数组既按行的方向又要按列的方向复制, 这就是一种嵌套的形式:

例 6.29 设  $A = \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$ , 则有  $\text{REPLICTE}(\text{REPLICATE}(A, \text{DIM}=2, \text{NCOPIES}=3), \text{DIM}=1, \text{NCOPIES}=2) \Rightarrow \begin{bmatrix} 11 & 12 & 11 & 12 & 11 & 12 \\ 21 & 22 & 21 & 22 & 21 & 22 \\ 11 & 12 & 11 & 12 & 11 & 12 \\ 21 & 22 & 21 & 22 & 21 & 22 \end{bmatrix}$

这时的复制规则是先执行内层的复制, 然后执行外层的复制。

## (2) 扩展函数 SPREAD

扩展函数的一般形式是:

$\text{SPREAD}(\text{SOURCE}, \text{DIM}, \text{NCOPIES})$

其中: SOURCE 可以是一个数值类型或者是一个逻辑类型的标量, 也可以是一个数值类型或者逻辑类型的数组(或数组片段)。DIM 是一个整数类型的标量表达式, DIM 取值范围与前述相同。当 DIM=1 时, 沿着数组第一维的下标变化的方向扩展, 也即向下扩展。当 DIM=2 时, 沿着第二维下标变化方向扩展即向右扩展。NCOPIES 是一个整型标量表达式, 即重复扩展的次数。

扩展函数功能: 把某数组中的一行(或全部数组)或一列拷贝若干次, 沿着行的方向或列的方向扩展, 构成一个新的数组。

例 6.30 当 SOURCE 是一个常量时, 扩展为一维数组。

$\text{SPREAD}(7, \text{DIM}=1, \text{NCOPILES}=5) \Rightarrow [7, 7, 7, 7, 7]$

$\text{SPREAD}(13, \text{DIM}=1, \text{NCOPILES}=-2) \Rightarrow [1:0]$

(注意: 这里  $[1:0]$  表示列大小为零的数组)。

例 6.31 当 SOURCE 是一维数组(或数组片段)时, 扩展为二维数组。

若一维数组 M 为  $M=[4, 2, 6, 3]$

则  $\text{SPREAD}(M, \text{DIM}=1, \text{NCOPILES}=3) \Rightarrow \begin{bmatrix} 4 & 2 & 6 & 3 \\ 4 & 2 & 6 & 3 \\ 4 & 2 & 6 & 3 \end{bmatrix}$

在这里, 复制是沿着第一维的下标复制三次变成  $3 \times 4$  的数组。

$\text{SPREAD}(M, \text{DIM}=2, \text{NCOPILES}=3) \Rightarrow \begin{bmatrix} 4 & 4 & 4 \\ 2 & 2 & 2 \\ 6 & 6 & 6 \\ -3 & 3 & 3 \end{bmatrix}$

此处复制过程是沿着第二维下标变化方向扩展三次,变成 $4 \times 3$ 数组。

例 6.32 设  $A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$

则  $\text{SPREAD}(A(2,2:4),1,3) \Rightarrow \begin{bmatrix} 22 & 23 & 24 \\ 22 & 23 & 24 \\ 22 & 23 & 24 \end{bmatrix}$

在这里  $\text{SPREAD}(A(2,2:4),1,3) = \text{SPREAD}(A(2,2:4),\text{DIM}=1,\text{NCOPIES}=3)$   
表示将 $[22,23,24]$ 重复 3 次,沿着第一维下标变化(即向下,行号改变)构成一个 $3 \times 3$ 新数组。

同理  $\text{SPREAD}(A(2,2:4),2,3) \Rightarrow \begin{bmatrix} 22 & 22 & 22 \\ 23 & 23 & 23 \\ 24 & 24 & 24 \end{bmatrix}$

此处沿着第二维下标变化(即向右扩展),重复 3 次构成一个新数组。

若当 SOURCE 是取一个二维数组进行,扩展将是一个三维数组。如:

$\text{SPREAD}(A,\text{DIM}=3,\text{NCOPIES}=2) \Rightarrow \begin{bmatrix} \begin{bmatrix} 11 & 12 & 13 & 14 \end{bmatrix} \\ \begin{bmatrix} 21 & 22 & 23 & 24 \end{bmatrix} \\ \begin{bmatrix} 31 & 32 & 33 & 34 \end{bmatrix} \end{bmatrix}$

此处是将二维数组重复 2 次,构成一个 $3 \times 4 \times 2$  的三维数组(注:请注意扩展函数 SPREAD 与复制函数 REPLICATE 的区别;扩展函数 SPREAD 将二维数组重复 2 次变成三维数组;而复制函数将二维数组重复 2 次后,仍然是二维数组)。

### § 6.3.5 重新整形函数

重新整形 RESHAPE 函数是按指定形状从 SOURCE 数组元素中生成一个新数组。当数组中还需要附加的元素来填充时,则由填充 PAD 项中的元素来补充。如果需要将 SOURCE 数组中的元素进行转置后存放到新数组中,则要增加可选次序 ORDER 项来表明需要转置,转置后元素在新数组中的存放顺序是以行将 SOURCE 中的元素进行存放。若 ORDER 项缺省时,SOURCE 数组中的元素在新数组中是以列顺序存放。

重新整形 RESHAPE 函数的一般形式是:

$\text{RESHAPE}(\text{MOLD}, \text{SOURCE}[, \text{PAD}][, \text{ORDER}])$

其中:MOLD 是一个整型的排列数组,它的大小范围从 1 到 7,数组中的每个元素必须是非负的。MOLD 的作用,规定重新整形后新数组模型的形状和大小,即规定数组元素的多少。SOURCE 是一个数值类型的数组或逻辑类型的数组,SOURCE 数组中的元素是生成一个新数组的基础。如果 PAD 项缺席或 PAD 项的大小是零,则 SOURCE 数组中元素的个数必须大于或等于模 MOLD 中元素的乘积。重新整形生成的新数组元素的个数,是 MOLD 中元素的乘积。可选项 PAD 是与 SOURCE 相同类型的数组,作用是为新生成数组提供填充的元素。PAD 中的元素可顺序地重复使用。可选项 ORDER 是一个整数类型的数组,它的形状

与模 MOLD 相同。它的值一定是[1:n]的一个转置，这里的 n 是模 MOLD 的大小。如果 ORDER 缺省，则预置的值为[1:n]。

**例 6.33** 假定需要构造一个  $3 \times 4$  的数组 A 中的各元素供应运行使用，即可使用重新整形。RESHAPE 函数构造出来。

A=RESHAPE(MOLD=[3,4],SOURCE=[1:12])

此处的模 MOLD 规定数组为 3 行 4 列，源数据为 1~12。存放数组 A 中是以列顺序存放。即

$$A = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

**例 6.34** 当提供源数据少于模 MOLD 规定元素时，应增加 PAD 项来填空位。如：

B=RESHAPE([3,4],SOURER=[1:5],PAD=[10:12])

此处的 MOLD=[3,4]，B 数组的存储方式如下：

$$B = \begin{bmatrix} 1 & 4 & 11 & 11 \\ 2 & 5 & 12 & 12 \\ 3 & 10 & 10 & 10 \end{bmatrix}$$

其中 PAD 中的元素 10~12 反复使用，直到将 B 数组填满为止。

又如 C=RESHAPE([4,5], $\begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix}$ ,PAD= $\begin{bmatrix} -1 & -4 \\ -2 & -5 \\ -3 & -6 \end{bmatrix}$ )

$$\text{则 } C = \begin{bmatrix} 1 & 5 & -1 & -5 & -3 \\ 2 & 6 & -2 & -6 & -4 \\ 3 & 7 & -3 & -1 & -5 \\ 4 & 8 & -4 & -2 & -6 \end{bmatrix}$$

**例 6.35** 将一个  $3 \times 4 \times 2$  的三维数组经过 RESHAPE 重新整形后生成  $6 \times 4$  的二维数组，如下所示：

若：

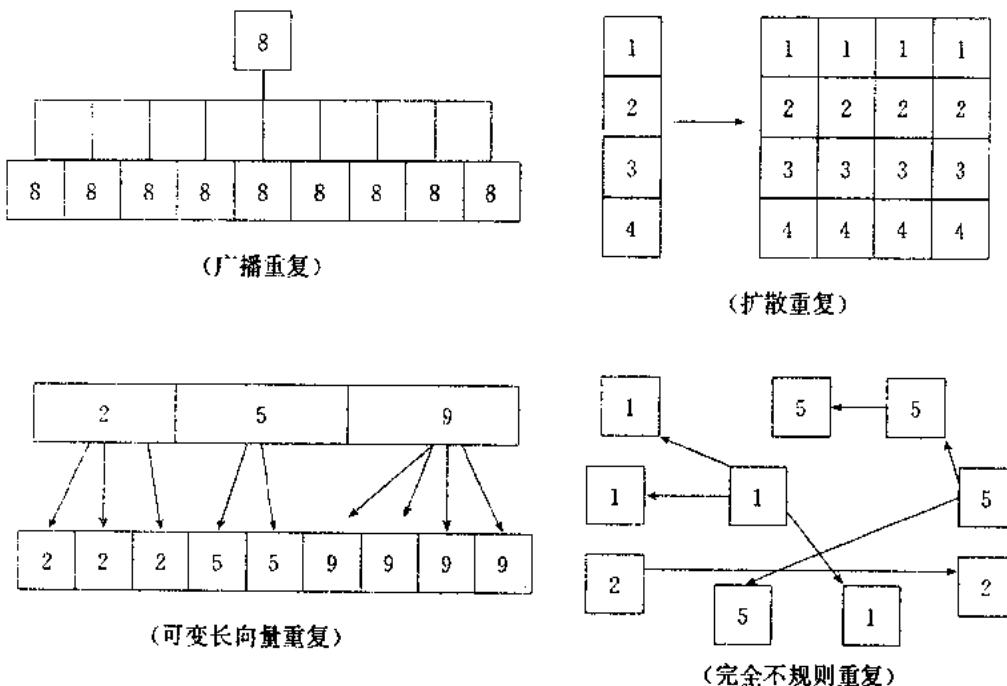
$$D = \begin{bmatrix} \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix} \\ \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix} \end{bmatrix}$$

则

RESHAPE([6,4],SOURCE=D,ORDER=[2,1])

$$\Rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix}$$

综上所述,数据的重复可简略如下图:



## § 6.4 向量点积和矩阵的乘法

矩阵乘法函数 MATMUL 操作在两个矩阵变元(也可以是一个向量)上,在矩阵乘法下返回的结果是两矩阵的乘积。向量点积 DOTPRODUCT 函数,执行后返回的结果是两向量的标量乘积。逻辑矩阵与向量,在矩阵乘法 MATMUL 下执行的是逻辑矩阵的乘法。而逻辑向量在向量点积 DOTPRODUCT 下执行,返回的结果是标量乘积。

### § 6.4.1 向量点积 DOTPRODUCT

向量点积 DOTPRODUCT 执行的数值或逻辑向量的点积乘法运算,返回的结果是标量。如果是逻辑类型的变元,点积 DOTPRODUCT 返回的结果就是逻辑类型结果。如果是数值类型的变元,点积 DOTPRODUCT 返回的结果是限定使用。限定使用规则如表 6.1。

向量点积 DOTPRODUCT 的一般形式为:

DOTPRODUCT(VECTOR\_A, VECTOR\_B)

其中 VECTOR\_A 是数值型或逻辑类型排列的数组; VECTOR\_B 要分两种情况,当 VECTOR\_A 是数值类型时,VECTOR\_B 就是一个数值类型的排列数组。如果 VECTOR\_A 是一个逻辑类型,则 VECTOR\_B 就是一个逻辑类型的排列数组。VECTOR\_B 的大小必须与 VECTOR\_A 的大小相同。

向量点积 DOTPRODUCT 执行后的结果分三种情况:

(1) 当 VECTOR\_A 是整型或实型时,返回后的结果值为:

SUM(VECTOR\_A \* VECTOR\_B)。

表 6.1

限定数值下标次序值

分类	显式形状规定	下标列表	下标次序值
1	$j_1:k_1$	$S_1$	$1 + (S_1 - j_1)$
2	$j_1:k_1, j_2:k_2$	$S_1, S_2$	$1 + (S_1 - j_1) + (S_2 - j_2) * d_1$
3	$j_1:k_1, j_2:k_2,$ $j_3:k_3$	$S_1, S_2, S_3$	$1 + (S_1 - j_1) + (S_2 - j_2) * d_1$ $+ (S_3 - j_3) * d_2 * d_1$
...	...	...	...
7	$j_1:k_1, j_2:k_2,$ $j_3:k_3, \dots, j_7:k_7$	$S_1, S_2, S_3,$ $\dots, S_7$	$1 + (S_1 - j_1) + (S_2 - j_2) * d_1$ $+ (S_3 - j_3) * d_2 * d_1 \dots$ $+ (S_7 - j_7) * d_6 * d_5 * \dots * d_1$

注意: 1.  $d_i = \text{MAX}(k_i - j_i + 1, 0)$ , 在这里  $d_i$  是数组的维数大小。  
 2. 如果数组大小是非零的, 则有  $j_i \leq S_i \leq k_i, i = 1, 2, \dots, 7$

如果向量的大小是零, 则返回结果值是零。其中, CONJG 函数的功能是对复数 VECTOR\_A 的虚部取相反符号。

(2) 当 VECTOR\_A 是复数类型, 返回后的结果值为:

SUM(CONJG(VECTOR\_A) \* VECTOR\_B)。

如果复数型向量的大小是零, 返回后的结果值也是零。

(3) 当 VECTOR\_A 是逻辑类型时, 返回后的结果值是:

ANY(VECTOR\_A . AND. VECTOR\_B)。

如果逻辑型向量的大小是零, 返回后的结果值是零・FALSE・。

向量点积举例如下:

例 6.39 DOTPRODUCT([1,2,3],[2,3,4]) $\Rightarrow$ 20

若 A=[1,2,3], B=[(2,0),(3,0),(-2,1)]

则 DOTPRODUCT(B,A) $\Rightarrow$ (2,-3)

DOTPRODUCT([T, · , T, · ], [ · , T, · , T]) $\Rightarrow$ F

DOTPRODUCT([T, · , · , T], [ · , T, · , T]) $\Rightarrow$ T

#### § 6.4.2 矩阵的乘法 MATMUL

矩阵的乘法 MATMUL 函数, 是执行数值或逻辑矩阵的乘法运算。如果是逻辑类型的变元, 矩阵乘法 MATMUL 返回的结果就是逻辑类型的结果。如果是数值类型的变元, 矩阵乘法返回的结果是限定使用。限定使用规则在表 6.1 中给出。

矩阵乘法 MATMUL 函数的一般形式为:

MATMUL(MATRIX\_A, MATRIX\_B)

其中 MATRIX\_A 是一列或两列的数值数组或逻辑数组。MATRIX\_B 要分两种情况, 如果 MATRIX\_A 是数值类型时, 则 MATRIX\_B 就是一个数值类型的数组。如果 MATRIX\_A 是逻辑类型, 则 MATRIX\_B 就是一个逻辑型的数组。如果 MATRIX\_A 是一维数组, 则

MATRIX\_B必定是二维数组;如果 MATRIX\_B 是一维数组,则 MATRIX\_A 必定是二维数组。矩阵乘法结果的大小维数是取第一个变元的行下标和取第二个变元的列下标组成一个数组的下标值。

矩阵乘法的结果形状,依赖于变元的形状。而变元形状又分如下三种情况:

① 如果 MATRIX\_A 的形状为[n,m]和 MATRIX\_B 的形状为[m,k],则结果的形状是[n,k]。

② 如果 MATRIX\_A 的形状为[m]和 MATRIX\_B 的形状为[m,k],则结果的形状是[k]。

③ 如果 MATRIX\_A 的形状为[n,m]和 MATRIX\_B 的形状为[m],则结果的形状是[n]。

矩阵乘法 MATMUL 执行的结果值也分三种情况:(1)如果矩阵乘法是数值类型的变元,结果 R 的元素为 R(i,j),则 R 等于:

SUM(MATRIX\_A(i,:) \* MATRIX\_B(:,j))

如果矩阵乘法是逻辑类型的变元,结果 R 的元素为 R(i,j),则 R 等于:

ANY(MATRIX\_A(i,:). AND. MATRIX\_B(:,j))

$$\text{例 6.36} \quad \textcircled{1} \quad \text{MATMUL}\left(\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}\right) \Rightarrow \begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$$

$$\textcircled{2} \quad \text{MATMUL}\left(\begin{bmatrix} T & T & \cdot \\ \cdot & T & \cdot \end{bmatrix}, \begin{bmatrix} T & T & \cdot & T \\ T & \cdot & T & \cdot \\ T & T & T & \cdot \end{bmatrix}\right) \Rightarrow \begin{bmatrix} T & T & T & T \\ T & \cdot & T & \cdot \end{bmatrix}$$

如果矩阵乘法是数值类型的变元,结果 R 的元素为 R(j),则 R 等于:

SUM(MATRIX\_A(:,j) \* MATRIX\_B(:,j))

如果矩阵乘法是逻辑类型的变元,结果 R 的元素为 R(j),则 R 等于:

ANY(MATRIX\_A(:,j). AND. MATRIX\_B(:,j))

$$\text{例 6.37} \quad \text{MATMUL}([1,2], \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}) \Rightarrow [5,8,11]$$

(3)如果矩阵乘法是数值类型的变元,结果 R 的元素为 R(j),则 R 等于:

SUM(MATRIX\_A(i,:)\* MATRIX\_B(:,j))

如果矩阵乘法是逻辑类型的变元,结果 R 的元素为 R(j),则 R 等于:

ANY(MATRIX\_A(i,:). AND. MATRIX\_B(:,j))

## § 6.5 数组应用实例

功能强大的数组变换处理的现代特性,为并行程序设计开辟了新的天地。并行程序融进现代特性,使得程序结构简洁、层次清楚,便于理解,易于设计,执行速度快。下面介绍运用数组的现代特性所设计的程序实例。

**例 6.38** 用高斯消去法解线性方程组(为讨论问题方便,取四阶线性方程组。对于任意阶问题,程序中只改一下变量即可)。

$$A: \begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = b_1 & (1) \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = b_2 & (2) \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = b_3 & (3) \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = b_4 & (4) \end{cases}$$

解：将四阶线性方程组的常数定义为  $A_{4 \times 4}$ ，常数项矩阵为  $B_{4 \times 1}$ ，另外再定义一个矩阵  $M_{4 \times 5}$ ，它的前四列即  $A$ ，第五列为  $B, M$  矩阵如下：

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & b_1 \\ a_{21} & a_{22} & a_{23} & a_{24} & b_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & b_3 \\ a_{41} & a_{42} & a_{43} & a_{44} & b_4 \end{bmatrix}$$

已知高斯消去法分两大步骤：消去，回代。第一步：通过适当的消去方法，使得所给方程组呈上三角形方程组  $\bar{A}$ ：

$$\bar{A}: \begin{cases} x_1 + \bar{a}_{12}x_2 + \bar{a}_{13}x_3 + \bar{a}_{14}x_4 = \bar{b}_1 & (1') \\ x_2 + \bar{a}_{23}x_3 + \bar{a}_{24}x_4 = \bar{b}_2 & (2') \\ x_3 + \bar{a}_{34}x_4 = \bar{b}_3 & (3') \\ x_4 = \bar{b}_4 & (4') \end{cases}$$

完成全部的消去过程，用通常的编程可表示为：

```

FOR—k:DO k=1,3
    B(k)=B(k)/A(k,k)
FOR—L:DOL=k,4
    A(k,L)=A(k,L)/A(k,k)
    END DO FOR—L
FOR—I:DOI=k+1,4
    B(I)=B(I)-A(I,k)*B(k)
FOR—J:DO J=k+1,4
    A(I,J)=A(I,J)-A(I,k)*A(k,J)
    END DO FOR—J
    END DO FOR—I
    END DO FOR—k

```

以上程序用了三重循环，而用数组构造函数完成全部消去过程，只要一层循环即可完成。程序段为：

```

DO k=1,3
    M(k,k+1:5)=M(k,k+1:5)/M(k,k)
    M(k,k)=1
    M(k+1:4,k+1:5)=M(k+1:4,k+1:5)
        -SPREAD(M(k,k+1:5),1,4-k)
        * SPREAD(M(k+1:4,k),2,4-k+1)
    M(k+1:4,k)=0

```

---

```
END DO
```

第二步是回代: 将(4')解得  $x_4$  代入(3')得解  $x_3$ , 将  $x_3$  代入(2')解得  $x_2$ , 其形式为:

$$\begin{cases} x_4 = \bar{b}_4 \\ x_3 = \bar{b}_3 - \bar{a}_{34}x_4 \\ x_2 = \bar{b}_2 - \bar{a}_{23}x_3 - \bar{a}_{24}x_4 \\ x_1 = \bar{b}_1 - \bar{a}_{12}x_2 - \bar{a}_{13}x_3 - \bar{a}_{14}x_4 \end{cases}$$

可用一般形式表示为

$$x(i) = B(i) - \sum_{j=i+1}^4 A(i,j) * x(j) \quad (i=3,2,1)$$

用通常的编程方法需要二重循环才能完成, 即程序段为:

$$x(4) = B(4)$$

```
FOR—I:DOI=3,1,-1
```

```
    s=0.0
```

```
FOR—J:DOJ=I+1,4
```

```
    s=s+A(I,J)*x(J)
```

```
    END DO FOR—J
```

```
    x(I)=B(I)-s
```

```
    END DO FOR—I
```

而用归约函数, 只要一层循环即可完成回代工作。下面给出的高斯消去法解线性方程组的完整程序: PROGRAM SOLRER—EQUATION

```
IMPLICIT NONE
REAL,DIMNSION(4,4)::A
REAL,DIMNSION(4)::x,B
REAL,DIMNSION(4,5)::M
INTEGER::k,I,J
READ *,((A(I,J))J=1,4),I=1,4),B
M(1:4,1:4)=A
M(1:4,5)=B
DO k=1,3
    M(k,k+1:5)=M(k,k+1:5)/M(k,k)
    M(k,k)=1
    M(k+1:4,k+1:5)=M(k+1:4,k+1:5)
        -SPREAD(M(k,k+1:5),1,4-k)
        * SPREAD(M(k+1:4,2,4-k+1)
    M(k+1:4,k)=0
END DO
DO k=4,1,-1
    x(k)=M(k,5)-SUM(M(k+1:4)*x(k+1:4))
END DO
```

```

RRINT * .x
END PEOGRAM SOLVER--EQUATION

```

例 6.39 考虑定常条件下粘性不可压缩流在二维有界区域  $\Omega$  上“空腔流动”满足 Navier-Stokes 涡度方程

$$\begin{cases} \Delta\psi = -\omega & \text{于 } \Omega \text{ 内} \\ \Delta\omega + R(\frac{\partial\psi}{\partial x}(\frac{\partial\omega}{\partial y} - \frac{\partial\psi}{\partial y}) - \frac{\partial\omega}{\partial x}) = 0 \end{cases} \quad (6.1)$$

其中  $\psi$  是流函数,  $\omega$  是涡度,  $\Delta$  为 Laplace 算子, 参数  $R$  称为 Retnolds 数, 为简单计, 设区域  $\Omega = \{(x, y) | 0 < x, y < 1\}$ , 它的边界记为  $\Gamma$ , 它以点  $A(0, 0), B(1, 0), C(1, 1), D(0, 1)$  为顶点, 流体在以  $\overline{DA}, \overline{AB}, \overline{BC}$  为内壁的容器里表面受到  $\overline{CD}$  方向的力的作用时的运动状态满足边界条件

$$\begin{cases} \psi = 0, \frac{\partial\psi}{\partial y} = -1 & \text{于 } \overline{DD} \text{ 上,} \\ \psi = 0, \frac{\partial\psi}{\partial n} = 0 & \text{于 } \Gamma \setminus \overline{CD} \text{ 上.} \end{cases} \quad (6.2)$$

其中  $n$  为  $\Gamma$  的外法线方向。

(6.1) 的算子方程形式为

$$LU = 0 \quad (6.3)$$

解(6.3)的 Newfon 法程序是:

$$L'[u_i](u_{i+1} - u_i) = -Lu_i \quad (6.4)$$

将(6.4)写成

$$\begin{cases} \Delta\psi_{i+1} + \omega_{i+1} = 0; \\ [\Delta + A(\psi_i)]\omega_{i+1} - A(\omega_i) \cdot \psi_{i+1} = -A(\omega_i)\psi_i \end{cases} \quad (6.5)$$

①解(6.5)式算法设计如下:

- 1) 将区域  $\overline{\Omega}$  分裂为 32 个互相重叠的矩阵子区域, 每个子区域(矩阵)的宽  $H = 1/32$ ;
- 2) 取步长  $h2 = (1/32)^2$ , 用五点差分算子将上述方程离散成代数方程组;
- 3) 采用红-黑迭代算法求解离散方程组, 取误差限  $me = 10^{-5}$  以控制迭代次数。

②在 CM-5 系统上运行的 CM Fortran 程序清单如下:

```

Program n-s.fcm
implicit none
include '/usr/include/cm/CMF_defs.h'
include '/usr/include/cm/timer-fort.h'
real * 8 beta
real * 4 trace
integer m,n,niter,nrec,recno
character * 1 answer
character * 4 line(20)
integer x,y,z,zz
real omg,step,r,omgl,omg2
parameter (z=31,zz=61)

```

```

real * 8, array(0:z,0:zz) :: F,U,UU,DF,a1,a2,b1,b2,w0,w1
real * 8, array(0:z,0:zz) :: w2,w3,w4
CMF $ layout F(,),U(,),UU(,),DF(,),a1(,),a2(,),b1(,)
CMF $ layout b2(,),w0(,),w1(,),w2(,),w3(,),w4(,)
logical 1 amask(0:z,0:zz),bmask(0:z,0:zz),cmask(0:z,0:zz),
         dmask(0:z,0:zz)
real re,me,error,h2
integer i,j,l,ii,iii
1  print *, 'x = ,y = ,omg = ,omg1 = ,omg2 = ,ii = iii = ,R = '
read *, x,y,omg,omg1,omg2,ii,iii,r
      step=1.0/x
open(unit=21,file='chen.dat',status='unknown')
nrec=0
90 format('x,y,ii,iii,omg,omg1,omg2:',4i4,3e9.5)
      print *, x,y,ii,iii,omg,omg1,omg2
      write(unit=21,fmt=90)x,y,ii,iii,omg,omg1,omg2

c start  timer for composite squaring
call cm_timer_clear(0)
call cm_timer_start(0)
h2=step * * 2
amask=.false.
bmask=.false.
cmask=.false.
dmask=.false.
forall (i=1:x-1,j=1:y-1,mod(i+j,2).eq.0) amask(i,j)=.true.
forall (i=1:x-1,j=1:y-1,mod(i+j,2).eq.1) bmask(i,j)=.true.
forall (i=2:x-2,j=2:y-2,mod(i+j,2).eq.0) cmask(i,j)=.true.
forall (i=2:x-2,j=2:y-2,mod(i+j,2).eq.1) dmask(i,j)=.true.
forall (i=1:x-1,j=1:y-1)cmask(i,j)=.true.
u=1.0
f=0.0
me=1.0
l=0
do while (me.gt.1.0e-5)
      l=l+1
      uu=u
      do i=1,ii
      forall(i=1:x-1)f(i,y-1)=f(i,y-2)*0.25-step*0.5

```

```

forall(i=1:x-1)f(i,1)=f(i,2) * 0.25
forall(i=2:y-2)f(x-1,i)=f(x-2,i) * 0.25
forall(i=2:y-2)f(1,i)=f(2,i) * 0.25
where (cmask)
  df=(cshift(f,1,1)+cshift(f,1,-1)
      +cshift(f,2,1)+cshift(f,2,-1)+u * h2) * 0.25-f
  f=f+omg * df
end where
where (dmask)
  df=(cshift(f,1,1)+cshift(f,1,-1)
      +cshift(f,2,1)+cshift(f,2,-1)+u * h2) * 0.25-f
  f=f+omg * df
end where
end do
forall (i=1:x-1)u(i,y)=(2.0 * (step-f(i,y-1))/h2-u(i,y)) * omg2+u(i,y)
forall (i=1:x-1)u(i,0)=(-2.0 * f(i,1)/h2-u(i,0)) * omg2+u(i,0)
forall (j=1:y-1)u(0,j)=(-2.0 * f(1,j)/h2-u(0,j)) * omg2+u(0,j)
forall (j=1:y-1)u(x,j)=(-2.0 * f(x-1,j)/h2-u(x,j)) * omg2+u(x,j)
a1=cshift(f,1,1)-cshift(f,1,-1)
b1=cshift(f,2,1)-cshift(f,2,-1)
a2=abs(a1)/2.0
b2=abs(b1)/2.0
w0=4.0+(r * a2+r * b2)
where(a1.ge.0.0)
  w2=1.0+a2 * r
  w4=1.0
elsewhere
  w2=1.0
  w4=1.0+a2 * r
end where
where(b1.ge.0.0)
  w1=1.0+b2 * r
  w3=1.0
elsewhere
  w1=1.0
  w3=1.0+b2 * r
end where
do i=1,iii
where(amask)

```

```

df=(w1*cshift(u,1,1)+w2*cshift(u,2,1)+w3*cshift(u,1,-1) +
w4*cshift(u,2,-1))/w0-u
u=u+omg*df
end where
where(bmask)
    df=(w1*cshift(u,1,1)+w2*cshift(u,2,1)+w3*cshift(u,1,-1) +
w4*cshift(u,2,-1))/w0-u
    u=u+omg*df
end where

end do
df=uu-u
re=sqrt(sum(df*df)/(x-1)/(y-1))
me=maxval(abs(df))
if (mod(1,10).eq.0) then
    write(6,* 1,re,me!,error
    write(unit=21,fmt=200) 1,re,me
200 format('Iteration',i9,'Average E',f10.7,'Max Error',f10.7)
    end if
end do
write(6,* ) 1,re,me
write(6,* ) x,omg,omg1,omg2,ii,iii
write(unit=21,fmt=201)x,omg,omg1,omg2,ii,iii,r,1,re
201 format('x=',i4,'omg=',f6.4,'omg1=',f6.4,'omg2=',f6.4,'ii=',i4,
        iii=i4,'r=',f6.2,'Iteration',i5,'Average E',f8.4)
call cm_timer_stop(0)
write(6,* ) 1,re,me
write(6,* ) x,omg,omg1,omg2,ii,iii
print 92,CM_timer_read_cm_busy(0),CM_timer_read_cm_idle(0)
92 format('times for N-S Eqs (busy,idle):',2F8.4)
write(unit=21,fmt=92)CM_timer_read_cm_busy(0),
CM_timer_read_cm_idle(0)
write(unit=21,fmt=125)
125 format('* * * * * * * * * * * * * * * * * * * * * * * * * * * *')
close(unit=21)
open(unit=21,file='nnn.dat',status='unknown')
40 format('x=',i3,'y=',i3,'ii=',i3,'iii',i3,3f5.3)
    write(21,41)
41 format('{')

```

```
do i=0,x
    write(21,42)(f(i,j),j=0,3)
42    format('(',4('f14.8,''))
    write(21,43)(f(i,j),j=4,y-4)
43    format(' ',4('f14.8,''))
    if(i.ne.x)then
        write(21,44)(f(i,j),j=y-3,y)
44    format(' ',3('f14.8,''),' ',f14.8,'')
        else
            write(21,45)(f(i,j),j=y-3,y)
45    format(' ',3('f14.8,''),' ',f14.8,'')
        end if
    end do
    write(21,46)
46    format('}')
    close(unit=21)
    stop
129   print 130
130   format(' $(C)ontinue or (S)top:_')
    read(*,131)answer
131   format(a1)
    if((answer.eq.'c').or.(answer.eq.'c'))go to 1
    if ((answer.ne.'s').and.(answer.ne.'s'))go to 129
    stop
end do
end
```



## 第三部分 并行程序通信

第七章 CMMMD 概述

第八章 CMMMD 同步通信函数

第九章 CMMMD 异步通信函数

第十章 CMMMD 应用实例

### 概要

本部分介绍连接计算机 (*Connection Machine*) CM-5 所装备的并行通信函数库 CMMMD 的一些主要函数功能及用法, 并通过对一个实例的详细介绍, 以期让读者获得 MIMD 程序设计的一些初步认识。



## 第七章 CMMD 概述

众所周知,一个并行计算机系统的通信能力是衡量其品质的重要尺度。目前,通信方式主要有两种:一种是传统的以点对点方式在相邻节点之间一对一地传送数据,另一种是一对多的广播方式,其特点是系统中每个节点都能“听”到其他节点发送的信息,而每个节点根据“呼叫地址”的选择决定是否接收这些信息。CM-5 计算机所装配的 CMMD 通信程序库包含这两种通信方式,限于篇幅,本书只介绍 CMMD 版本 3.0 中通信程序的机理及用法。

CMMD 是一个装配在 CM-5 系统上的信息传递例行程序库,一个用 CMMD 函数进行通信的并行程序具有如下两个特点:

- CM-5 系统的每一个节点运行一个子程序,并且管理与本身有关的计算和数据;
- 节点之间的通信是通过调用 CMMD 函数实现的。

CMMD 允许从一个节点到另一个节点用多种不同的方式发送信息,具体采用何种方式取决于并行程序的需要。CMMD 提供点对点方式和广播方式的信息传递函数等。本书只对这两种函数中常用的予以介绍,其他内容读者可参考有关书籍。

CMMD 函数库可以被由 C、C++、Fortran 77 和 CM Fortran 等语言编写的程序调用。

### § 7.1 程序模型

CMMD(版本 3.0)支持两个程序模型:

- 主/节点(host/node)程序模型。它包含两个同时运行的程序:一个是作为主程序运行在主机上。首先它执行必要的初始化(包括开始的 CMMD 通信环境),然后启动各个节点上运行的程序。另一个作为子程序运行在各个节点上,它们都是同一个程序的拷贝,换言之,即各个节点上运行的是同一个程序,它们通常是求子问题的程序,当然,各个节点执行程序所涉及的数据因节点而异,它与该节点所承担的子问题有关。

- 无主机(hostless)程序模型。在这种模型中,使用主机仅仅是启动和结束整个执行过程以及作输入输出服务。这一输入输出服务程度由 CMMD 提供。而运行在各节点的程序则与前一模型所叙情况完全一致。根据需要,节点之间可进行通信。

在 CMMD(版本 3.0)中,无主机模型的程序可以用 C、Fortran 77 和 CM Fortran 等语言编写。主/节点模型的程序必须有用某种串行语言编写的主程序,但其可以调用 C\* 和 CM Fortran 等并行语言编写的模块。

### § 7.2 通信协议

对于一般的用户,CMMD 提供遵守“相互交换信息”的同步和异步通信协议。这种同步是指相匹配的发送信息的节点与接收信息的节点之间的同步,而在其他时间,每个节点的计算相对于其他节点都是异步的。CMMD 也允许完全异步的通信。诚如人们所知道的,异步通

信通常是间断性的,一个节点发出一个准备发送或者接收信息的信号,然后继续另外的工作直到其相关节点(准备接收该节点发送的信息或者发送信息给该节点的节点)已经作好了交换信息的准备,但若是具有优先权的信息,则可以通过轮询来驱动异步信息传递。

为了优化信息传递的重复模式的性能,CMMD 提供了虚拟通道。利用虚拟通道,两个节点之间建立一条可以反复使用的传输链路,通道一旦建立,发送节点写一个预定义的数组到通道,而接收节点读该通道,然后置零以作他用,不需要同步。

每个节点都拥有少量任何另外节点中具有代表性的数据的必要信息(除了整个系统所有节点共用的函数和数据之外),这些信息主要是指数据的类型、大小和存放的位置,为了建立两个节点之间的一条通道,需要初级的“相互交换信息”步骤,在这一步骤中,节点相互交换将要传递的数据的类型和大小、源节点(发送信息的节点)和目标节点(接收信息的节点)等信息。

下面我们对在同步或者异步中点对点的通信方式作稍微深入的探讨。

在点对点的信息传送中,信息的源节点和目标节点在交换中取同等的地位。一个发送信息的节点在信息发送中会完成如下四个步骤:

- ①发出要发送数据的意图给将要接收数据的节点;
- ②指定将要接受数据的节点;
- ③在通信期间,为数据提供一个“用户标识符”即数据的标号;
- ④指定要发送的数据存放在本节点内存的什么地方。

而接受信息的节点在此期间则会完成:

- ①登入一个接收缓冲区以便接收数据;
- ②指定将要接收信息的节点;
- ③指定“用户标识符”,以便能准确找到标明了的信息。

一旦接收数据的节点和发送数据的节点双方发出了已作好准备的信号,数据交换立即进行。值得指出的是,两个节点或者可能等待全部交换操作完成后再进行接下来的工作(即同步传送),或者他们各自完成本身在交换中的操作后,便重新开始已被中断了的交换之前的工作,而不顾及相关节点的交换操作是否已经完成,换言之,交换以异步方式进行。

顺便提及,广播方式的通信与上述在很多方面类似,不同之处是前者需要所有节点立即参与整个通信操作,每个节点必须做到:

- ①发送参与整个通信操作的信号;
- ②在存储区内指定何处可以找到将发送出去的数据(或者存放将要接收的数据)。

广播通信只能在参与的所有节点同步操作的情况下完成。

下面,我们简要介绍一些在信息中使用的数组类型,由 CMMD 传送的数据块其代表形式是数组,它取自下面三种类型:

- ①串行数组——存放在单个节点存储器内的一个数据块,串行数组也称为块串行数组;
- ②向量串行数组(向量数组)——将一个存储在单个节点存储器内的数组,按照两个元素之间以固定的距离展开后的数组;
- ③并行数组——存储在一个单个节点的向量处理单元中的数组。

### § 7.3 CMMID 的输入输出

信息传送程序需要灵活的输入输出。一个并行程序希望有如下功能：

- 一个节点读或者写文件；
- 所有节点读或者写文件，且每一个节点都是独立于其他节点；
- 所有节点用某些协议的方式读或者写文件。

CMMID 推广了 UNIX I/O，其协议提出了独立地输入输出。如果文件是对所有节点打开，则它可以是下面三种方式之一：

- ①在独立方式，任何一个节点可以独立地读或者写文件；
- ②在同步-顺序方式，所有节点同时读和写文件，每次读或写文件的一段（但是连续的）；
- ③在同步-广播方式，文件的一段被同时读和广播到所有节点。仅仅是 0 号节点可以对该文件做写操作。

CMMID 也提供双精度文件指针，允许用户程序访问非常大的文件。有关输入输出的详细内容请阅本书所列有关参考文献。

### § 7.4 CM-5 的体系结构

联结计算机(Connection Machine)CM-5 是美国思维机器公司(Thinking Machines Corporation)生产的大型并行计算机系统(参见附录 4)，CM-5 的设计者解放思想，大胆采用通用的系统结构，把 SIMD 和 MIMD 机器的优点结合在一起。按照惯例，超级计算机的程序员往往要在 MIMD 计算机和 SIMD 计算机之间作出选择。MIMD 机器的优点是善于处理独立的转移，但是在同步和通信方面存在不少问题，相比之下，SIMD 机器的同步通信功能很强，但转移处理的能力较差，而 CM-5 设计成同步 MIMD 结构可同时支持两种并行计算方式，它包含若干数量(通常是 32, 64, 128)的节点，若干个 I/O 控制处理机(IOCPs)和 I/O 设备，少量的主机。这些组件通过两个内部通信网-数据和控制网连结在一起(见图 7-1)。

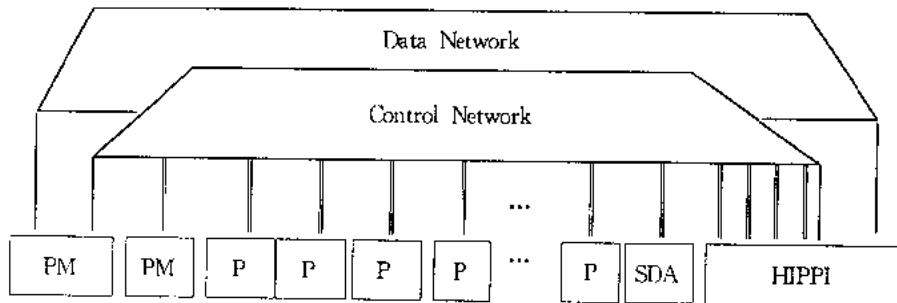


图7-1 CM-5系统

使用数据网的目的是为了节点之间、主机之间和 I/O 设备之间数据进行快速和高带宽的交换，而使用控制网则是为了提供诸如广播通信、查询操作和节点间的同步等服务。

每一个节点包含一个 RISC 微处理器和一个网络接口芯片，同时还包含 4 个安置在 RISC 微处理器与节点存储器之间的向量处理单元(VUs)，其目的是提供基于存储器的高效

算术运算(见图 7-2)。

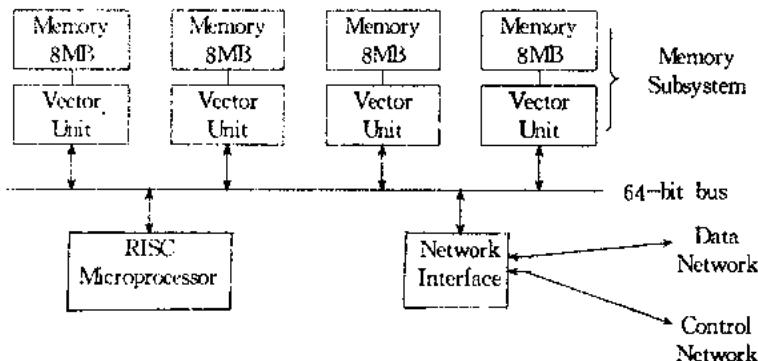


图 7-2 一个 CM 节点的组成图

CM-5 的节点被分类成一个或多个分区,每一个分区由一个管理单元控制。程序总是在管理单元上开始执行,在管理单元上结束执行。CM-5 系统使用一个并行分时操作系统 CMOST,它允许每一个分区像一个独立的并行分时系统一样运行,每一个分区用它自己的进程集合去操作,甚至可以当做是一个独立的并行处理系统,从而节省了与其他分区和 I/O 设备的通信。

## § 7.5 一个简单的 CMMD 程序

下面是一个用 C 语言编写的 CMMD 程序,它是一个单独的程序,被复制后运行在 CM-5 系统所有节点(可选择 32,64 或 128 等数目)上。

```
# include<stdio.h>
# include<cmmd.h>
main(argc,argv)
    int argc;
    char * argv [];
{
    CMMD_fset_io_mode.Stdout,
    CMMD_sync_seq();
    print("Hello world,from node %d\n",CMMD_self_address())
}
```

该程序被执行后,在用户终端上将显示如下结果:

```
>hello world
Hello world,from node 0
Hello world,from node 1
Hello world,from node 2
...
```

显示结果的数目取决于 CM-5 系统中节点的数目。函数 CMMD\_fset\_io\_mode 是以同步-顺序方式执行标准的输出,所以显示的结果按照节点的次序,从 0 到最高的编号数。上面的程

序也可以改写为只让部分结点显示结果,例如改写上面程度的第六、七、八行为:

```
CMMMD_fset_io_mode(Stdout,CMMMD_independent);  
if(CMMMD_self_address()<10)  
    print("Hello world,from node %d.\n",CMMMD_self_address());
```

请注意后一程序的参数 CMMMD\_independent,这是独立输入方式的标志,它允许某些指定的节点显示结果,而前面程序的参数 CMMMD\_sync\_seq 是同步-顺序方式的标志,它需要所有节点都显示结果。

## 第八章 CMMD 同步通信函数

本章介绍同步形式的点对点的 CMMD 通信函数。这些通信函数主要有

CMMD\_send\_block{..V}  
 CMMD\_receive\_block{..V}  
 CMMD\_send\_and\_receive{..V}  
 CMMD\_swap

发送数据和接收数据的节点，在发送数据之前先须保持同步。一旦一个节点调用了上述函数中的某一个，那么它则将信息分块，或者等待去共享那些已经完成了传输的信息。更详细地说，发送节点一直要待到将要发送的信息全部送进了数据网，而接收节点则要待到将来自于数据网的全部信息接纳进了内存，方可执行其他操作。

这些函数可以与其他的同步通信函数混合使用，亦可与异步通信函数混合使用。例如，一条信息可以用一个异步通信函数发送，而用一个同步通信函数接收。

整个函数名可分为三部分：大写的 CMMD 是所有通信函数的标志（关键字）；花括号中的“\_V”是可缺省项，它是否缺省取决于下面的情况：若函数所发送（接收）的信息是向量时，必须写“\_V”，若信息是非向量时则不写“\_V”；CMMD 与“\_V”之间的部分是表示该函数功能的关键字，函数既可带有参数也可以无参数。有参数时参数填写在函数后面的圆括号内。

### § 8.1 缓冲区和数组

通过通信函数传递的信息也就是数组或者缓冲区。大多数同步通信使用两类数组：串行数组和并行数组。需要指出，异步通信函数不接收串行数组。下面简要介绍这两类数组。

串行数组即微处理存储器组，它们驻在节点或者主机的微处理器存储器内。它们可以是 C 语言和 FORTRAN 语言的变量，CM Fortran 的“FE arrays”。CMMD 函数的串行数组是由存储单元（缓冲区自变量）和长度(buffer\_desc 自变量) 定义。串行数组分为非向量串行数组和向量串行数组。

并行数组驻在节点的 VU 存储器内而不能在主机内。在 CM Fortran 中，一个并行数组代表一个 CM 数组。当在通信函数中使用一个 CM 数组时，如果用 CMMD\_PARALLEL\_ARRAY 关键词作为 buffer\_desc 自变量，那么 CMMD 函数得到所有关于该数组的类型、大小等信息。

### § 8.2 发送消息函数

发送一条信息的 CMMD 函数有两个，其一般形式为

```
CMMD_send_block;
CMMD_send_block_V.
```

这两个函数在 Fortran 语言和 C 语言中的形式和规则大同小异,故我们将只在 Fortran 语言中的形式和规则予以叙述(在以后的章节中我们都采取这一原则)。

其 Fortran 形式为

```
INTEGER FUNCTION CMMD_send_block(dest_node,tag,buffer,buffer_desc)
integer dest_node,tag,buffer_desc<array>buffer
INTEGER FUNCTION CMMD_send_block_V(dest_node,tag,buffer,elem_len,
stride,elem_count)
integer desc_node,tag,elem_len,stride,elem_count
<array>buffer
```

下面从三个方面来详细讨论这两个函数。

### 1. 自变量说明

`dest_node`:标识接收信息的目标节点的整数;  
`tag`:为发送信息设置的一个标号,或者为 CMMD\_DEFAULT\_TAG。显然,它是一个正整数;

`buffer`:存放被发送信息的缓冲区;

`buffer_desc`:缓冲区描述符。对于串行数组,须用一个非负整数说明该缓冲区的长度(位数)。对于 CM Fortran 并行数组,则用关键字 CMMD\_PARALLEL.. ARRAY 说明(仅限于发送向量信息时使用)。

`elem_len`:说明向量中每一个元素、长度和整数(仅限于发送向量信息时使用)。

`stride`:说明向量中诸元素与起始位置之间距离(位数)。它要么是 `elem_len` 的倍数,要么是零(仅限于发送向量信息时使用)。

`elem_count`:说明向量中元素数量的一个整数(仅限于发送向量信息时使用)。

### 2. 关于这两个函数机理的一些说明

函数 CMMD\_send\_block 发送指令的缓冲区的内容到给定的目标节点(长度为 0 的缓冲区则发送长度为 0 的信息),该节点必须在本分区内,否则给出错误结果。信息可以由用户提供的标识符来辨认;符号 CMMD\_DEFAULT\_TAG 是标准的缺省标识符。直到接收信息的节点已经调用了 CMMD 中的一个接收函数,并且给出了已作好准备接收信息的明确表示,即发出某种信息,传送才会开始。

需要指出,串行数组必须发送给串行缓冲区,并行数组必须发送给并行缓冲区。如果想在串行数组和并行数组之间进行数据传送,则必须分两个步骤进行:首先对串行数组和并行数组进行变换,然后再进行传送。

在以后的章节中,会经常用到 handler 函数这一名词。所谓 handler 函数是指用户自编的函数过程,一个 handler 函数可以执行计算、同步或异步通信。

本节所介绍的两个函数既不能用 handler 函数,也不能被 handler 函数所调用。

在以后的章节中,会经常讨论 handler 函数与该节所介绍的函数的关系。

### 3. 返回值说明

上述函数待信息传递完毕后,返回如下值:

如果返回值为 0,说明所有的信息都被发送出去;  
 如果返回值为 1,则说明不是所有的数组元素都被发送出去;  
 若返回信息为 CMMD\_ERRVAL,则说明发送过程中出现了一个错误。

### § 8.3 接收信息函数

接收一条信息的 CMMD 函数有两个,其一般形式为

CMMD\_receive\_block

CMMD\_receive\_block\_V

其 Fortran 形式为

```
INTEGER FUNCTION CMMD_receive_block (source_node, tag, buffer, buffer_desc)
```

```
integer source_node,tag,buffer_desc
```

```
<array>buffer
```

```
INTEGER FUNCTION CMMD_receive_block_V (source_node, tag, buffer, elem_len,stride,elem_count)
```

```
integer source_node,tag,elem_len,stride,elem_count
```

```
<array>buffer
```

#### 1. 自变量说明

source\_node: 标识发送信息的源节点的整数(CMMD\_ANY\_NODE 允许任何一个节点都可以发送信息);

tag: 作为被发送信息标号的一个正整数,或者是 CMMD\_DEFAULT\_TAG,如果是 CMMD\_ANY\_TAG,则允许接收用任何一个标识符作标号的信息;

buffer: 存放被接收信息的缓冲区;

buffer\_desc: 缓冲区描述符,定义接收信息的缓冲区位数的一个非负整数,特别是,对于一个 CM Fortran 并行数组,则用关键字 CMMD\_PARALLEL\_ARRAY 说明(仅限于发送非向量信息);

elem\_len: 说明向量中每个元素长度(位数)的一个整数(仅限于向量函数);

strids: 说明向量中诸元素与起始位置之间距离(位数),要么是 elem\_count 的倍数,要么是 0(仅限于向量函数);

elem\_count: 说明向量中元素数量的一个整数(仅限于向量函数)。

#### 2. 关于这两个函数机理的一些说明

函数 CMMD\_receive\_block 首先登入一个缓冲区,然后等待接收一条带有指定的标识符的信息,这条信息由指定的源节点所发送。这些程序可以采用特殊的符号 CMMD\_ANY\_NODE 作为源自变量,表明任何源节点所发送的信息都是可接收的,并以符号 CMMD\_ANY\_TAG 作为标识符自变量,表明任何标识符都将被接收。

当利用 CMMD\_send\_block 程序时,其缓冲区不能与用在通信函数中的其他缓冲区相重叠。并行数组必须在发送节点和接收节点上指明其大小。

上述两个函数既不能调用 handler 函数,也不能被 handler 函数所调用。

### 3. 返回值说明

当发送来的信息已经被接收而且已经被拷贝到指定的缓冲区后, 上述函数返回一个整数值, 具体情况如下:

如果返回值为 0, 说明传送成功, 所有的信息均被接收;

如果返回值为 1, 说明不是所有的数组元素被接收, 接收失败;

如果返回信息为 CMMD\_ERRVAL, 则说明接收过程中出现一个错误。

## § 8.4 同时发送和接收函数

其一般形式为

```
FUNCTIONS CMMD_send_and_receive
          CMMD_send_and_receive_V
```

它的 Fortran 形式为

```
INTEGER FUNCTIONS CMMD_send_and_receive ( source_node, source_tag,
inbuffer,inbuf_desc,dest_node,dest_tag,outbuffer,outbuf_desc )
```

```
integer source_node,source_tag,inbuf_desc,
integer inbuf_desc,dest_node,
integer dest_node,desctag,outbuf_desc,
<array>inbuffer,outbuffer
```

```
INTEGER FUNCTIONS CMMD_send_and_receive_V ( source_node, source_tag,
inbuffer, in_elem_len, in_stride, in_elem_count, dest_node, dest_tag, outbuffer, out_
elem_len,out_stride,out_elem_count )
```

```
integer source_node,source_tag
integer in_elem_len,in_stride
integer in_elem_count,dest_node
integer dest_tag,out_elem_len
integer out_stride,out_elem_count
<array>inbuffer,outbuffer
```

### 1. 自变量说明

source\_node: 标识源节点的一个整数;

source\_tag: 用来作为被接收的信息标号的一个正整数, 或者是 CMMD\_DEFAULT\_TAG, 或者是 CMMD\_ANY\_TAG;

inbuffer: 将存放被接收信息的缓冲区;

inbuffer\_desc: 将存放由节点所接收的信息的缓冲区的长度(位数)。对于 CM Fortran 并行数组, 则用关键字 CMMD\_PARALLEL\_ARRAY 说明(仅限于非向量函数);

in\_elem\_len: 由该节点所接收的向量中每个元素的长度(位数), 应该是相关数据类数量的一个倍数(仅限于向量函数);

in\_stride: 该节点接收的信息所组成的向量中诸元素与起始位置之间的距离(位数),

要么是 in\_elem\_len 的倍数, 要么等于0(仅限于向量函数);

in\_elem\_count: 由该节点所接收的信息组成的向量中元素的总数(仅限于向量函数);

dest node: 标识目标节点的一个整数;

dest tag: 作为由该节点发送的信息标号的一个正整数。

outbuffer: 存放由该节点发送的信息的缓冲区。

outbuf\_desc: outbuffer 的长度(位数), 对于 CM Fortran 并行数组, 则用关键字 CMMD\_PARALLEL\_ARRAY 说明(仅限于非向量函数);

out\_elem\_len: 由该节点发送的向量中每个元素的长度(位数), 应该是一个合适的数

---

```
integer out_elem_len,out_stride
integer out_elem_count
<array>inbuffer,outbuffer
```

### 1. 自变量说明

partner: 用来标识两个进行信息交换的节点;

inbuffer: 该缓冲区用来存放被接收的信息;

inbuf\_desc: 上面 inbuffer 的长度(位数),或者对于 CM Fortran 并行数组,则用关键字 CMMD\_PARALLEL\_ARRAY 说明(仅限于 CMMD\_swap)。

in\_elem\_len: 由该节点所接收的向量中每个元素的长度(位数),应该是一个相关数据类数量的倍数(仅限于 CMMD\_swap\_V);

in\_stride: 由该节点所接收的向量中诸元素与起始位置之间的距离(位数),要么是 in\_elem\_len 的一个倍数,要么是0(仅限于 CMMD\_swap\_V)。

in\_elem\_count: 该节点所接收的向量中元素的总数(仅限于 CMMD\_swap\_V);

outbuffer: 存放由该节点发送的信息的缓冲区;

outbuffer\_desc: 对于串行数组,它表示将存放由该节点所接收的信息缓冲区的长度,对于 CM Fortran 并行数组,则用关键字 CMMD\_PARALLEL\_ARRAY 说明(仅限于 CMMD\_swap);

out\_elem\_len: 由该节点所发送的向量中每个元素的长度(位数),应该是相关数据类数量的一个倍数(仅限于 CMMD\_swap\_V);

out\_stride: 被发送的向量中诸元素与起始位置之间的距离(位数),应该是 out\_elem\_len 的一个倍数或者是0(仅限 CMMD\_swap\_V);

out\_elem\_count: 该节点发送的信息所组成的向量中元素的总数(仅限于向量函数)。

### 2. 关于函数机理的一些说明

CMMD\_swap 与 CMMD\_swap\_and\_receive 功能完全一样。同样,CMMD\_swap\_V 与 CMMD\_swap\_and\_receive\_V 其功能也完全一样。这里,源节点和目标节点是等价的。注意,用户不要为交换操作而说明某些标识符,事实上,交换函数会自动用 CMMD\_DEFAULT\_TAG 标识自己。

3. 返回值情况与上节类似,不再赘述。

## § 8.6 节点信息函数(辅助函数)

在信息输送期间,每台处理机都必须清楚地知道相互的地址,而寻址过程则是通过利用节点标识符实现,它们可以被看做是节点的逻辑地址。对于每一个分区,节点标识符集合包含着从0到该分区内的节点总数减1之间的所有整数。例如若该分区内有128个节点,则节点标识符取0,1,…,127。

节点可以调用 CMMD\_partition\_size 函数以测定当前分区的大小,而且可以调用 CMMD\_self\_address 以测定它本身在分区内的位置。

下面分别来介绍上述两个函数。

## 1. FUNCTION CMMD\_self\_address

其 Fortran 形式为

```
INTEGER FUNCTION CMMD_self_address()
```

运行在一个给定节点上的进程调用此函数,便可得该节点的标识符(编号)。

## 2. FUNCTION CMMD\_partition\_size

其 Fortran 形式为

```
INTEGER FUNCTION CMMD_partition_size()
```

任何一个节点调用此函数,便可得到当前分区中节点的数量。

例:若打算对一个向量作若干循环移位,而该向量分布在全部节点上,则可编程如下:

```
if(CMMD_self_address() < (CMMD_partition_size() - 1))
    right_neighbor = CMMD_self_address() + 1;
else
    right_neighbor = 0;
if(CMMD_self_address() > 0)
    left_neighbor = CMMD_self_address() - 1;
else
    left_neighbor = CMMD_partition_size() - 1;
```

## § 8.7 信息检测函数

其一般形式为

```
FUNCTION CMMD_msg_pending
```

其 Fortran 形式为

```
INTEGER FUNCTION CMMD_msg_pending(node, tag)
```

## 1. 自变量说明

node: 节点标识符(编号)

(可以是 CMMD\_ANY\_NODE)

tag: 信息标号(可以是 CMMD\_ANY\_TAG)

## 2. 函数机理的一些说明

该函数检测是否存在一个等待被接受的信息。如果返回值是1,则说明有一个有用的信息处于等待状态,否则返回0。任何一个节点可以调用该函数以检测是否存在一条等待被接收的信息。

## § 8.8 信息存取器函数

CMMD 提供了四个与同步信息传送函数一起使用的存取器函数,它们将告知你如下一些信息:发送和接收了多少信息?由什么节点发送?标号是什么?

这四个函数是:

CMMD\_bytes\_received

CMMD\_bytes\_sent

CMMD\_msg\_sender

CMMD\_msg\_tag

这些函数可以在程序执行期间和信息传送期间的任何时刻调用。具体地说，当一个信息检测函数(前一节介绍的 CMMD 函数)返回一个 1 时，调用上述函数则可确定实际发送的信息有多少。亦可得到来自于 CMMD\_ANY\_NODE 或者以 CMMD\_ANY\_TAG 标识的信息的有关数据。

上述函数的返回值通过各自调用 CMMD\_msg\_pending 和信息检测函数得以修正。

下面我们分别予以介绍

#### 1. FUNCTIONS CMMD\_bytes\_received

CMMD\_bytes\_ent

其 Fortran 形式为

INTEGER FUNCTIONS CMMD\_bytes\_received()

INTEGER FUNCTIONS CMMD\_bytes\_sent()

CMMD\_bytes\_received 返回由调用节点接收的信息的位数，这个信息不是被接收的最近的信息，就是由 CMMD\_msg\_pending 检测过的信息。

CMMD\_bytes\_sent 返回由调用节点发送的最后一条信息的位数。

#### 2. FUNCTIONS CMMD\_msg\_sender

CMMD\_msg\_ag

其 Fortran 形式为

INTEGER FUNCTIONS CMMD\_msg\_sender()

INTEGER FUNCTIONS CMMD\_msg\_tag()

这两个函数分别返回节点标识符或者由调用节点接收的最后一条信息的标号。若该节点调用了 CMMD\_msg\_pending，且返回值是 TRUE，则调用 CMMD\_msg\_sender 以返回等待发送一条信息的那个节点的标识符，而调用 CMMD\_msg\_tag 则返回所等待发送信息的标号；若该节点调用 CMMD\_msg\_pending 且返回值是 FALSE，则调用上述两个函数分别返回最后节点的标识符或者信息的标号。

## 第九章 CMMD 异步通信函数

本章介绍异步形式的点对点的 CMMD 通信函数。这些通信函数主要是：

CMMD\_send\_async  
CMMD\_receive\_async  
CMMD\_send\_noblock

这些函数允许一个节点发出准备传送一条信息的信号，然后继续其他的操作，直到某一个节点响应了这一要求为止。当两者都已准备好之后，则中断正在进行的操作，信息在此时被发送和接收。

两个异步发送函数在怎样处理数据上方式有所不同：CMMD\_send\_async 在传送期间发送缓冲区内的任何内容；而如果 CMMD\_send\_noblock 不能立即发送它的信息，它便把这些数据存储在一个临时缓冲区内，然后发送其缓冲区的内容。

需要指出的是，上述三个异步 CMMD 函数均不接收向量串行数组。CMMD\_send\_async 和 CMMD\_receive\_async 仅仅接收完全的并行数组，CMMD\_send\_noblock 不接收并行数组。

一个使用异步信息的程序必须确保：该程序返回或者存在之前，所有的信息已经发送和接收。由于这一原因，因此该程序结束之前，应该调用 CMMD\_all\_msgs\_wait 函数或者 CMMD\_all\_msgs\_done 函数，以保证它们在后来得到信息“TRUE”。

### § 9.1 异步发送函数

FUNCTION CMMD\_send\_async

其 Fortran 形式为

```
INTEGER FUNCTION CMMD_send_aasync (dest_node, tag, buffer, buffer_desc,
handler, handler_arg)
    integer dest_node, tag, buffer_desc
    <array>buffer
    SUBROUTINE handler(mcb, handler_arg)
        integer mcb
        <array>handler_arg
    SUBROUTINE handler(mcb, handler_arg)
        integer mcb
        <array>handler_arg
```

#### 1. 自变量说明

dest\_node: 目标节点;  
 tag: 信息用户标号,可以是任何32位的正整数,或者是CMMD\_DEFAULT\_TAG;  
 buffer: 信息的起始位置;  
 buffer\_desc: 对于串行数组,它说明被发送信息位数的一个非负整数。对于 CM  
                   Fortran 并行数组,则用关键字CMMD\_PARALLEL\_ARRAY 说明。  
 buffer(mcb,handler\_arg): 一个用户提供的函数(或者为空)。它的两个自变量说明  
                   如下:

mcb: 在 Fortran 形式中, 表示 mcb(信息控制块)的标识符;  
handler\_arg: 传送给 handler 函数的用户自定义数据结构。如果不需要, 则应是空或者是0。

## 2. 函数机理的一些说明

该函数发送已说明的缓冲区中的内容到给定的目标节点，该缓冲区具有指定的标号。标号可以是 CMMD\_DEFAULT\_TAG。当这一函数已经把发送的信息排队后，便返回一个信息控制块的标识符。显然，它几乎是立即有返回值。那么，用户程序应有如下两个职责：一是必须维持缓冲区的正确状态直到信息确实已经被发送和接收；二是当信息控制块 MCB 不再需要时，必须释放 CMMD\_mcb（即当信息已经被发送和接收后）。

程序可以定义一个 handler 函数，以备当信息发送之后引用。为了对一个专门的信息定义一个 handler 函数，给出 handler 的名字作为 CMMD\_send\_async 的参数，则当信息被发送时将引用 handler。

如果程序不提供 handler 函数，它仍然可以询问有关 MCB 的信息和释放 MCB，不过需要用到有关的辅助函数（稍后将作介绍）。

handler 函数一般设有两个自变量：一个是关于 MCB 的指示字，一个是用户提供的 handler\_arg。

### 3. 返回值说明

若返回值是0或者正整数，则说明 MCB 到了本次传送的末尾。

若为 CMMDF\_ERRVAL，则说明本次传送没有 MCBs。

## § 9.2 异步接收函数

## FUNCTIONS CMMI\_receive\_async

其 Fortran 形式为

```

INTEGER FUNCTIONS CMMD_receive_async (source_node, tag, buffer, buffer_
desc, handler, mcb, hamdler_arg)
    integer source_node, tag, buffer_desc
    <array>buffer
SUBROUTINE handler(mcb,handler_arg)
    integer mcb
    integer handler_arg .
SUBROUTINE handler(mcb,handler_arg)

```

```
integer mcb
<array> handler_arg
```

### 1. 自变量说明

source\_node: 源节点, 亦可以是 CMMD\_ANY\_NODE;  
 tag: 被接受信息的用户标号。可以是任何正整数, 或者是 CMMD\_DEFAULT\_TAG 或者是 CMMD\_ANY\_TAG;  
 buffer: 将存放被接收信息的缓冲区;  
 buffer\_desc: 对于串行数组, 它是说明被发送信息位数的一个非负整数。对于 CM Fortran 并行数组, 则用关键字 CMMD\_PARALLEL\_ARRAY 说明。  
 (handler)(mcb, handler\_arg): 一个用户提供的函数。其参数说明如下:  
     mcb: 在 Fortran 形式中, 表示它本身的标识符。  
     handler\_arg: 由 handler 函数所用的用户自定义数据结构。如果不需要, 则该自变量应该是空或者是 0。

### 2. 函数机理的一些说明

CMMD\_receive\_async 登入一个缓冲区以存放接收的信息, 而且立即返回一个信息控制块的指示字。

用户程序可以调用 MCB 询问函数(稍后作介绍)以测定信息何时已被接收。

注意, 如果对 CMMD\_send\_async 或者 CMMD\_receive\_async 的调用结果被分配到一个整体变量, 则不能保证在 handler 函数被调用之前会完成赋值。

如果接收信息的节点不知道发送给它的信息的长度, 它可以说明一个长度足够大的缓冲区去接受发送来的信息。该缓冲区不得与其他正在使用的缓冲区重叠。

### 3. 返回值说明

该函数一旦建立了一个描述被接收信息的 MCB, 就立即返回 MCB 的一个指示字, 返回值具体情况如下:

若返回值为 0 或者是正整数, 说明 MCB 到了这次传送的末尾。

若返回 CMMD\_ERRVAL, 说明本次传送没有 MCBs。

## § 9.3 非块化发送函数

FUNCTIONS CMMD\_send\_noblock

其 Fortran 形式为

```
INTEGER FUNCTIONS CMMD_send_noblock(dest_node, tag, buffer, buffer_desc)
    integer dest_node, tag, buffer_desc
    <array> buffer
```

### 1. 自变量说明

dest\_node: 目标节点;  
 tag: 作为信息标号的一个正整数, 或者为 CMMD\_DEFAULT\_TAG;  
 buffer: 存放被发送信息的缓冲区;  
 buffer\_desc: 由串行数组发送的信息的位数。

### 2. 函数机理的一些说明

该函数发送一个长度和标号均被规定了的串行缓冲区的内容到给定的目标节点, 该节点必须在本分区内, 否则给出一个错误结果。函数不能处理并行数组或者向量串行数组。

如果 CMMD\_send\_noblock 不能立即发送信息, 它便拷贝这些数据到一个临时缓冲区, 对用于后来的缓冲区进行排队, 而且给出返回值, 然后 CMMD 等待一个作好了接收信息准备的节点发出信号, 这时, 它中断发送节点的工作, 以便该节点可传送缓冲区中的信息。

该函数既不能调用 handler 函数, 也不能被 handler 函数所调用。

### 3. 返回值说明

如果返回值为 0, 说明信息被全部传送或者把后来发送的信息拷贝到了一个临时缓冲区。

若返回 CMMD\_ERRBAL, 则说明出现了一个错误。

## § 9.4 异步检测函数

函数 CMMD\_mcb\_pending 检测是否有一个等待被接收的信息。如它检测到了一个需要接收的信息, 便建立且返回一个 MCB, 任何一个节点都可以调用该函数去检测一条信息是否是自己所需要的, 然后调用 receive\_block 或者 receive\_sync 函数以获取等待被发送的信息。

其一般形式为:

FUNCTIONS CMMD\_mcb\_pending

其 Fortran 语法为

INTEGER FUNCTIONS CMMD\_mcb\_pending(node,tag)

integer node,tag

### 1. 自变量说明

node: 节点编号(也可以是 CMMD\_ANY\_NODE)

tag: 表示标号的整数(也可以是 CMMD\_ANY\_TAG)

### 2. 函数机理的一些说明

如果 CMMD\_ANY\_NODE 被说明, 则函数 CMMD\_mcb\_node 可以被调用以得到发送那个等待被接收的信息的节点的标识符。如果 CMMD\_ANY\_TAG 被说明, CMMD\_mcb\_tag 返回那个等待被接收的信息的标号。

该函数可以被 handler 函数调用。

### 3. 返回值

该函数返回那个等待被接收的信息的 MCB, 如果没有信息处于等待, 则返回 NULL。

## § 9.5 MCB 存取器函数

通过调用异步函数 CMMD\_send\_async 和 CMMD\_receive\_async, 可得到返回的 MCBS, 即信息控制块, 那么下面的这些函数可用于从 MCBS 中摘录信息。

FUNCTIONS CMMD\_mcb\_node

```
CMMD_mcb_source
CMMD_mcb_dest
CMMD_mcb_tag
CMMD_mcb_bytes
CMMD_mcb_buffer
```

其 Fortran 形式为

```
INTEGER FUNCTIONS CMMD_mcb_node(mcb)
INTEGER FUNCTIONS CMMD_mcb_source(mcb)
INTEGER FUNCTIONS CMMD_mcb_dest(mcb)
INTEGER FUNCTIONS CMMD_mcb_tag(mcb)
INTEGER FUNCTIONS CMMD_mcb_bytes(mcb)
<array> FUNCTIONS CMMD_mcb_buffer(mcb)
    integer mcb
```

自变量 MCB 表示一个信息控制块标识符。这些函数返回该信息的有关数据，内容如下：

CMMD\_mcb\_node： 返回相应节点的标识符，它可以被发送节点或接收节点调用；  
 CMMD\_mcb\_source： 返回源节点的标识符，该函数仅能为接收节点调用；  
 CMMD\_mcb\_dest： 返回目标节点的标识符，它只能为发送节点所调用；  
 CMMD\_mcb\_tag： 返回指定信息的用户标号；  
 CMMD\_mcb\_bytes： 返回指定信息中的数据位数；  
 CMMD\_mcb\_buffer： 返回用于发送(或接收)信息的缓冲区。  
 这些函数都可以被 handler 函数所调用。

## § 9.6 释放信息控制块子程序

```
FUNCTIONS CMMD_free_mcb
```

其 Fortran 形式为

```
SUBROUTINE CMMD_free_mcb(mcb)
integer mcb
```

自变量 mcb 是对 CMMD\_send\_async 或者 CMMD\_receive\_async 的调用而返回的信息控制块的标识符(ID)。信息一旦被发送和接收，则该函数允许指定的信息控制块立即释放。该函数没有返回值。可以被 handler 函数调用。

## § 9.7 等待异步信息子程序

本节所讨论的函数可以被用于使某个节点等待，直到全部信息已经传送完毕。

```
FUNCTIONS CMMD_msg_wait
```

```
CMMD_all_msgs_wait
```

其 Fortran 形式为

```
SUBROUTINE CMMD_msg_wait(mcb)
    integer mcb
SUBROUTINE CMMD_all_msgs_wait()
```

第一个子程序的自变量 mcb 是一个信息控制块标识符。这两个子程序都是用于使一个已经完成了异步信息传送的节点处于一种同步状态。具体地说, CMMD\_msg\_wait 强制调用节点对信息进行封装, 或者等待一个专门的信息被传送完毕。而 CMMD\_all\_msgs\_wait 则导致节点对信息进行封装, 直到所有的信息结束等待状态。这两个子程序都可以被 handler 函数调用。

## § 9.8 节点广播函数

```
FUNCTIONS CMMD_bc_to_nodes
    CMMD_receive_bc_from_node
```

其 Fortran 形式为

```
SUBROUTINE CMMD_bc_to_nodes(buffer,buffer_desc)
    <array>buffer
    integer buffer_desc
SUBROUTINE CMMD_receive_bc_from_node(buffer,buffer_desc)
    <array>buffer
    integer buffer_desc
```

### 1. 自变量说明

buffer: 存放将被广播出去的信息的缓冲区。在其他节点, 则是接收该信息的缓冲区。

buffer\_desc: 上述缓冲区的长度。

### 2. 关于函数机理的一些说明

任何一个节点都可以调用 CMMD\_bc\_to\_node 子程序以广播一个指定的缓冲区到其他节点(即接收这一缓冲区中信息的所有节点), 而所有其他节点则必须调用 CMMD\_receive\_bc\_from\_node 子程序以接收该缓冲区中的信息, 当然, 必须保证对应自变量的长度相等。不言而喻, 当上述操作结束后, 所有节点都具有该缓冲区中的信息。如果所有节点的长度自变量不相等, 则可能产生分段错误。另外, 所有在本分区内的处理机都必须加入这一操作, 否则, 操作将失败, 而且程序可能挂起。广播函数既不能调用 handler 函数也不能被 handler 函数调用。

### 3. 返回值说明

广播函数不返回任何值。

## 第十章 CMMD 应用实例

本章将详细介绍作者在 CM-5 并行计算机上求解一个数学问题的程序。目的是通过这个程序使读者对 CMMD 的一些基本函数的应用有一个初步的印象。

### § 10.1 例题及其算法

选择的例题是具有代表性的一类弱非线性边值问题

$$\begin{cases} \Delta u = u^2 & \text{in } \Omega = \{(x, y) | 0 < x < 1, 0 < y < 1\} \\ u|_{\Gamma} = \varphi \end{cases} \quad (10.1)$$

其中  $\varphi$  为大于等于 -1 的常数。 $\Delta$  为 laplace 算子

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

按照参考文献[1]中所建立的区域分裂法的框架, 我们把  $\Omega$  分裂成  $m=32$  个相互重叠的子区域(参见图10-1)

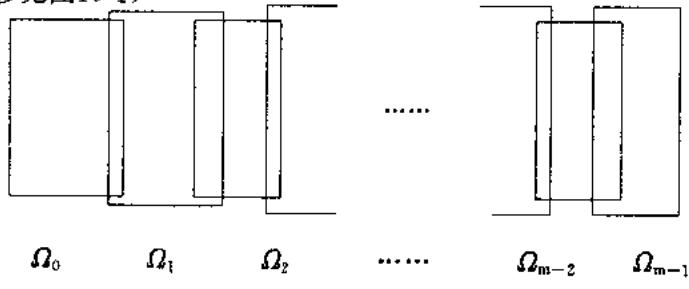


图10-1

问题(10.1)也分解成分别与  $m$  个子区域相结应的  $m$  个子问题的集合:

$$\begin{cases} \Delta u_j = u_j^2 & \text{in } \Omega_j \\ u_j|_{r_j} = \varphi_j \\ u_j|_{r_{j,j-1}} = u_{j-1} \\ u_j|_{r_{j,j+1}} = u_{j+1} & j = 0, 1, 2, \dots, m-1 \end{cases} \quad (10.2)$$

其中  $r_{j,j-1}$  为  $\Omega_j$  的左拟边界,  $r_{j,j+1}$  为  $\Omega_j$  的右拟边界。

(10.2) 的每个子问题是一个非线性问题, 我们采用 Picard 线性化方法把它们化成线性问题, 然后用文献[1]中所建立的 S-COR (Schwarz-Chaotic Over Relaxation) 算法去求解。即构成所谓 P-S-COR 算法, 它可用形式语言描述如下:

Algorithm P-S-COR

$W := \varphi$

```

E:=1
For j=1 to m do in parallel
begin
L:if E>e then
begin
u:=W
for i=1 to L do
begin
v:={v | Δv(i) = (vk(i))^2} in Ωj
u:=u+ωx(v-u)
end
εj = ||W-u on Ωj ||
W:=u
(E:=(Σεi2)1/2 if j=1)
go to L
end
stop
end

```

其中  $\Delta v_j(i) = (v_j^k(i))^2$  表示(10.2)中第  $j$  个 Picard 线性化方程。 $k$  为外迭代(Picard 线性化迭代)次数。

文献[1]证明了这一算法的收敛性。

我们对  $\Omega$  采用步长 = 1/256 的均匀正方形网格部分, 用五点差分格式离散子问题集(10.2)。则  $\Omega$  的 32 个子区域具有如下形式:

$$\begin{aligned}\bar{\Omega}_0 &= \{(ih, jh) | 0 \leq i \leq \lambda + d, 0 \leq j \leq 256\} \\ \bar{\Omega}_l &= \{(ih, jh) | l\lambda + (l-1)d + 1 \leq i \leq (l+1)(\lambda + d), 0 \leq j \leq 256\} \\ &\quad l = 1, 2, \dots, 30.\end{aligned}$$

$$\bar{\Omega}_{31} = \{(ih, jh) | 31\lambda + 30d + 1 \leq i \leq 256 + d, 0 \leq j \leq 256\}$$

其中  $\lambda = 3$  为每个子区域在  $x$  轴上非重叠的网点个数,  $d = 5$  为相邻两个子区域在  $x$  轴上重叠的网点个数。

(10.2) 的离散形式为

$$\begin{cases} (u_{i,j+1}^L + u_{i-j,j}^L + u_{i,j+1}^L + u_{i,j-1}^L - 4u_{i,j}^L)/h^2 = (u_{i,j}^L)^2 & \text{in } \Omega_L \\ u_{i,j}^L|_{\Gamma_j} = \varphi_j \\ u_{i,j}^L|_{\Gamma_{j,j-1}} = u_{i,j}^{L-1} \\ u_{i,j}^L|_{\Gamma_{j,j+1}} = u_{i,j}^{L+1} \\ L = 0, 1, 2, \dots, 31 \end{cases} \quad (10.3)$$

用红-黑迭代法求解(10.3)。将(10.3)中的 32 个子问题分别由 CM-5 并行计算机的 32 个节点去计算, 各节点之间进行必要的通信。此即为 Picard-Schwarz 型的异步并行算法。

## § 10.2 程序及其说明

按照 § 10.1 所阐述的算法, 编制了一些在 CM-5 并行计算机上运行的 CM Fortran 程序(源程序见附录1)。上节中用形式语言书写的 Algorithm P-S-COR 显示了源程序的基本结构。

下面对附录1中的第一个源程序作一些必要的解释, 涉及到 CMMD 函数的语句将尽可能详细。

程序采用无主机(hostless)模式。即程序的开始和结束, 以及输入输出等任务由主机承担, 在本程序中, 0号节点除了分配给本身的计算之外, 还担任了主机的全部工作。

整个程序大体分为三部分: 第一部分为主程序, 第二部分为子程序, 承担了全部主要的计算工作, 第三部分是两个 handler 子程序。

第一部分中几个主要的变量意义如下:

$x$ :  $\Omega$  在  $x$  轴上的网点总数, 包括两个端点;

$self$ : 节点标识符(编号);

$C$ : 节点个数(32);

$iii$ : 相邻两个子区域重叠网点个数。

$ii$ : 内迭代次数(即 Algorithm P-S-COR 中的内循环次数  $L$ )

$Xiii(3)$ : 具有三个常数的数组:

$$\begin{cases} Xiii(1)=x \\ Xiii(2)=iii \\ Xiii(3)=ii \end{cases}$$

$omg$ : 求解离散方程集合(10.3)的红-黑迭代法有松弛因子;

$omg1$ : 拟边界松弛因子;

$yom(2)$ : 具有两个常数的数组

$$\begin{cases} yom(1)=omg \\ yom(2)=omg1 \end{cases}$$

这一部分是主程序, 负责程序的开始、结束和输入输出。由0号节点担任。

首先, 调用两个 CMMD 节点信息函数, 分别得到0号节点的编号和划分的节点数( $C=32$ )。接着调用 CMMD 安全指令函数, 即调用进行诊断错误的函数。紧接着调用两个打开5号和6号输入输出通道的 CMMD 函数。

如果是第0号节点, 则调用两个 CMMD 广播通信发送函数将  $Xiii$  和  $yom$  两数组广播出去。如果是非0号节点, 则调用两个 CMMD 广播通信接收函数接收  $Xiii$  和  $yom$ 。

最后调用两个打开文件的 CMMD 函数。

第二部分是子程序  $ddm$ , 它被拷贝成32份分配给32个节点, 即每个节点求解一个编号与其相同的子区域上的(§ 10.3)中的子问题。

这一部分的主要功能是用红-黑迭代法求解离散的子问题与其相邻的两个子区域( $\Omega_0$ 只有右邻接子区域,  $\Omega_{31}$ 只有左邻接子区域)进行通信。不言而喻, 实际上也是节点与其编号邻接的节点之间的通信。

ddm 中的几个主要变量说明如下：

a,b： 分别为子区域的左端点和右端点(对整个区域亦然)；

df： 离散方程相邻两次迭代值的差；

ru,Lu： 分别为该子区域各重叠区网点值和左重叠区网点值；

me： 存放各子区域迭代值误差的一维数组。显然它有32个分量；

mm：存放子区域迭代值误差的变量；

ee：存放 me 的某种范数，即整个区域迭代值误差的某种范数；

amask,bmask： 在红-黑迭代中引入的两个屏蔽矩阵(逻辑矩阵)。

程序首先赋初值和处理边界条件(子区域)，然后进入外层循环(即 Picard 线性化迭代)。外层循环直到屏蔽语句 where(cmask) 出现之前结束。

按照前节所叙述的算法，在此循环中，首先用红-黑迭代法求解(§ 10.3)中的分配给该节点的子问题，迭代进行 ii 次，松弛因子为 omg。迭代结束后计算出迭代误差。

如果该节点的编号大于等于1，即该节点非0号节点，则调用 CMMMD 异步发送函数将误差发送给0号节点。

如果是0号节点，则做如下几件事：首先调用 CMMMD 检测函数，检测是否有等待接收的消息，如果有——即 b 不等于0，那么调用 CMMMD\_mcb\_node 函数取得发送该信息的那个节点的编号，然后调用 CMMMD 异步接收函数，接收该信息。待全部信息接收完毕后(即 DO while 循环结束)求出 me 的最大值并调用广播函数将该最大值广播给所有非0号节点。

如果是非0号节点，则调用广播接收函数接收0号节点广播来的信息(用来决定外层循环是否结束)。在外层循环结束之前，或者说进行下一轮外层循环之前，所有节点实际上是所有子区域，都必须与其相邻的左子区域和右子区域交换重叠区的信息，具体地说就是：如果是编号大于0的节点，调用 CMMMD\_swap 函数，则将本节点计算出来的 lu 与左邻接子区域上计算出来的 ru 进行交换，如果是编号小于31的节点，调用 CMMMD\_swap 函数，则将本节点计算出来的 ru 与右邻接子区域上计算出来的 lu 进行交换。这种交换是为下一轮 Schwarz 型的迭代提供新的初始值。

## 附录10.1 CM Fortran 源程序

```
C      u2.FCM(non-linear)
      implicit none
      integer x, self, a, b, c, iii, d, ii, xiii(3)
      real omg, omgl, yom(2)
      include "/usr/include/cm/cmmmd_fort.h"
      self=cmmmd_self_address()
      c=cmmmd_partition_size()
      call CMMMD_enable_safety()
      d=CMMMD_Set_io_mode(6,CMMMD_independent)
      d=CMMMD_Set_io_mode(5,CMMMD_independent)
      d=0
```

```

if (self.eq.0) then
  write(6,*)'x,xii,ii:'
  read(5,* )xiii(1),xiii(2),xiii(3)
  call CMMD_bc_to_nodes(xiii,12)
  write(6,*)'omg,omg1:'
  read(5,* )yom(1),yom(2)
  d=1
  call CMMD_bc_to_nodes(yom,8)
else
do while(d.eq.0)
end do
call CMMD_receive_bc_from_node(xiii,12)
call CMMD_receive_bc_from_node(yom,8)
end if
x=xiii(1)
iii=xiii(2)
ii=xiii(3)
omg=yom(1)
omg1=yom(2)
a=self*x/c-iii
b=(self+1)*x/c+iii
if (self.eq.0) a=0
if (self.eq.c-1) b=x
d=CMMD_Set_io_mode(0,CMMD_independent)
d=CMMD_Set_open_mode(CMMD_sync_seq)
end

subroutine ddm(x,a,b,c,d,self,iii,ii,omg,omg1)
implicit none
integer x,a,b,c,d,self,ii,iii,g,h,i,j,l,mcb
real, array(a:b,0:x):: F,u,uu,df,uuu
real, array(0:x):: ru,lu,rb,lb
logical, array(a:b,0:x):: amask,bmask,cmask
real step,h2,omg,omg1,ee,me(0:c-1),mm
real re,nn,e,gg, hh,umax,umin
integer sd(0:c),rv(0:c),ss(0:c),tt(0:c)
external handler1,handler2
include "/usr/include/cm/cmmd_fort.h"
step=1.0/x

```

```

forall (i=0:c) sd(i)=i
forall(i=0:c) rv(i)=c+i+1
gg=99
hh=88
me=1.0
h2=step ** 2
amask=.false.
bmask=.false.
cmask=.false.
cmask(a+1:b-1,1:x-1)=.true.
forall(i=a+1:b-1,j=1:x-1,mod(i+j,2).eq.0)amask(i,j)=.true.
forall(i=a+1:b-1,j=1:x-1,mod(i+j,2).eq.1)bmask(i,j)=.true.
uu=0.0
u=1.0
forall(i=a:b)u(i,0)=-100.0
forall(i=a:b)u(i,x)=-100.0
if(a.eq.0)forall(j=1:x-1)u(a,j)=-100.0
if(b.eq.x)forall(j=1:x-1)u(b,j)=-1.0
if(self.eq.31)write(0,*)'Au=u^2,u=-100,-100,-100,-1'
ee=1.0
l=0
do while (ee.gt.1.0e-4)
    l=l+1
    uuu=u
    do i=1,ii
        df=0.
        where(amask)
            df=(cshift(u,1,1)+cshift(u,1,-1)+cshift(u,2,1)+cshift(u,2,-1))/(
                4.0+h2*u)-u
        u=u+omg * df
    end where
    where(bmask)
        df=(cshift(u,1,1)+cshift(u,1,-1)+cshift(u,2,1)+cshift(u,2,-1))/(4.0
            +h2*u)-u
        u=u+omg * df
    end where
end do
re=sqrt(sum(df*df)/(b-a-1)/(x-1))
df=u-uuu

```

```

me(self)=maxval(abs(df))
mm=me(self)
if(self.ge.1)then
  h=CMMMD_send_async(0,CMMMD_DEFAULT_TAG,
    mm,4,handler1,sd(self),tt)
  if(h.eq.CMMMD_ERRVAL) write(0,*)'send error1'
end if
if (self.eq.0)then
  i=0
  h=CMMMD_mcb_pending(CMMMD_ANY_NODE,g)
  if(h.ne.0)then
    ss(i)=CMMMD_mcb_node(h)
    h=CMMMD_receive_async(ss(i),CMMMD_DEFAULT_TAG,nn,4,handler2,rv
      (i),tt)
    if (h.eq.CMMMD_ERRVAL) write(0,*)'send error2'
    me(ss(i))=nn
    i=i+1
  end if
  ee=maxval(me)
  call CMMMD_bc_to_nodes(ee,4)
end if
if(self.gt.0)call CMMMD_receive_bc_from_node(ee,4)
if((self.eq.0).and.(mod(1,100).eq.0))write(0,*)
1'self,1,re,ee',self,1,re,ee!,error
  forall(i=0:x) lu(i)=u(a+2*iii,i)
  forall(i=1:x) ru(i)=u(b-2*iii,i)
  if(self.gt.0) then
    h=-1
    h = CMMMD_swap (self - 1, ru, CMMMD_PARALLEL_ARRAY, lu, CMMMD_
PARALLEL_ARRAY)
  if(h.eq.CMMMD_ERRVAL) write(0,*)'send error3'
  if(h.eq.0)forall (i=0:x)u(a,i)=(ru(i)-u(a,i))*omg1+u(a,i)
end if
  forall(i=0:x) lu(i)=u(a+2*iii,i)
  forall(i=1:x)ru(i)=u(b-2*iii,i)
  if(self.lt.c-1) then
    h=-1
    h = CMMMD_swap (self + 1, lu, CMMMD_PARALLEL_ARRAY, ru, CMMMD_
PARALLEL_ARRAY)

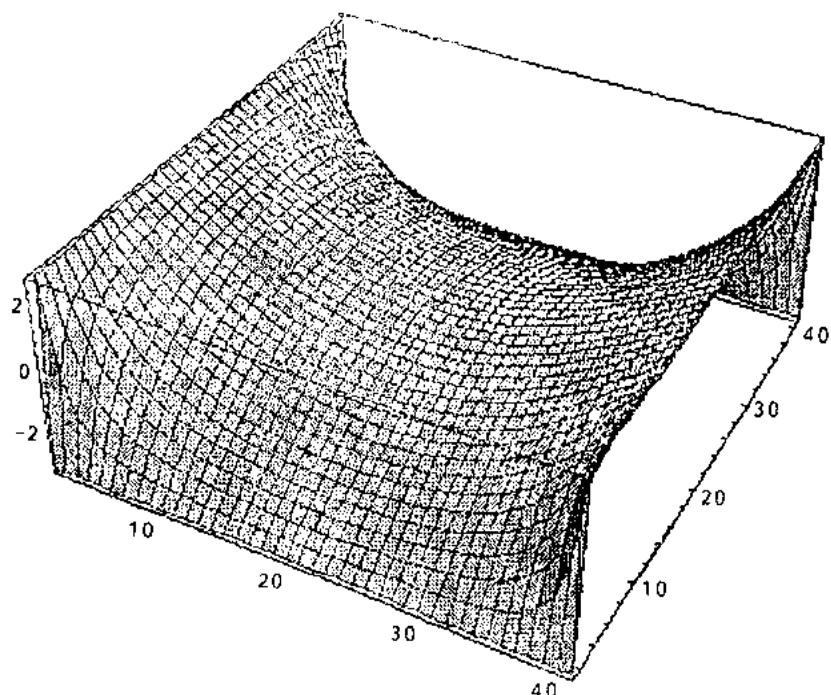
```

```
if(h.eq.CMMD_ERRVAL)write(0,*)'send error4'
  if(h.eq.0)forall (i=0:x)u(b,i)=(lu(i)-u(b,i))*omg1+u(b,i)
end if
end do
where (cmask) uu=u-uu
umax=maxval(u,mask=cmask)
umin=minval(u,mask=cmask)
if(self.eq.31)write(0,*)'Iterations=',l
write(0,*)'self,re,umax,umin=',self,re,umax,umin
if(self.eq.0)then
  write(0,*)'x,omg,omg1,iii,ii',x,omg,omg1,iii,ii
  write(0,100) u
format(4e12.4)
end if
return
end

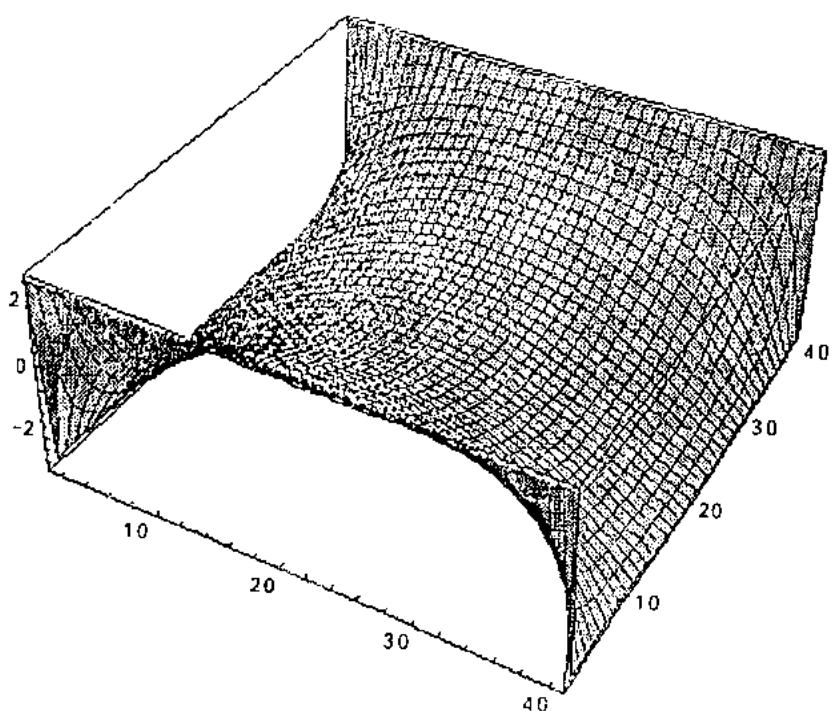
subroutine handler1(mcb,tt)
integer mcb,tt(33)
include "/usr/include/cm/cmmmd_fort.h"
call CMMD_free_mcb(mcb)
return
end

subroutine handler2(mcb,tt)
integer mcb,tt(33)
include "/usr/include/cm/cmmmd_fort.h"
call CMMD_free_mcb(mcb)
return
end
```

## 附录10.2 CM Fortran 源程序数值结果的图形显示



I 边界条件连续时的图形



II 边界条件不连续时的图形

图 10-1 M1 相控阵天线剖面图

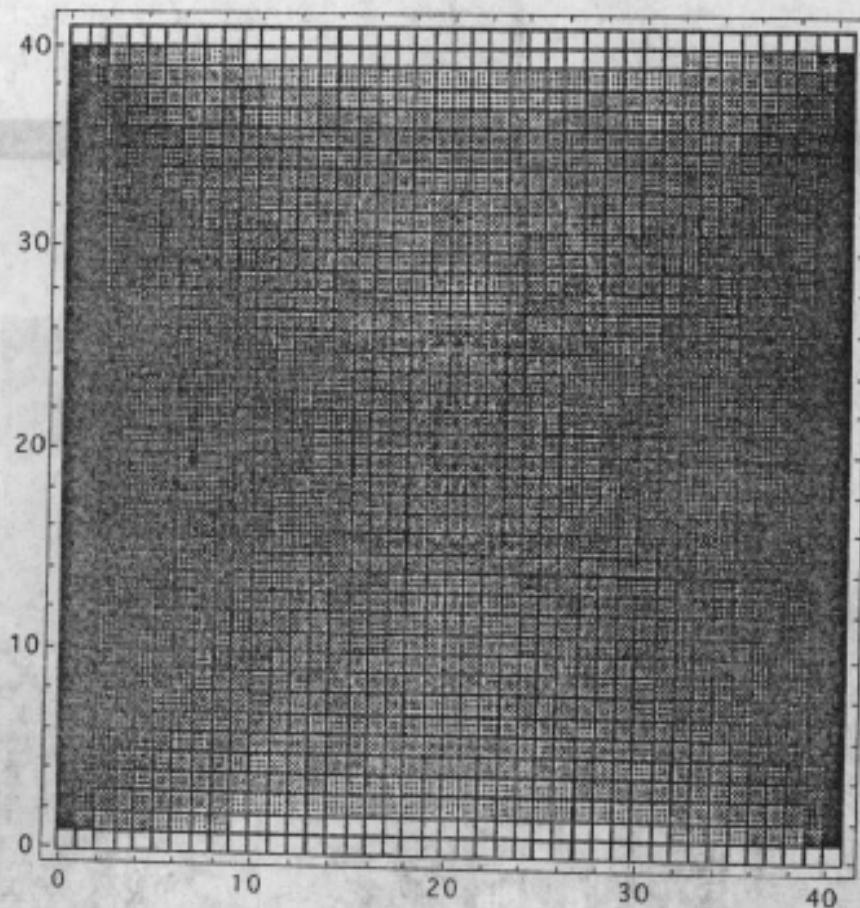
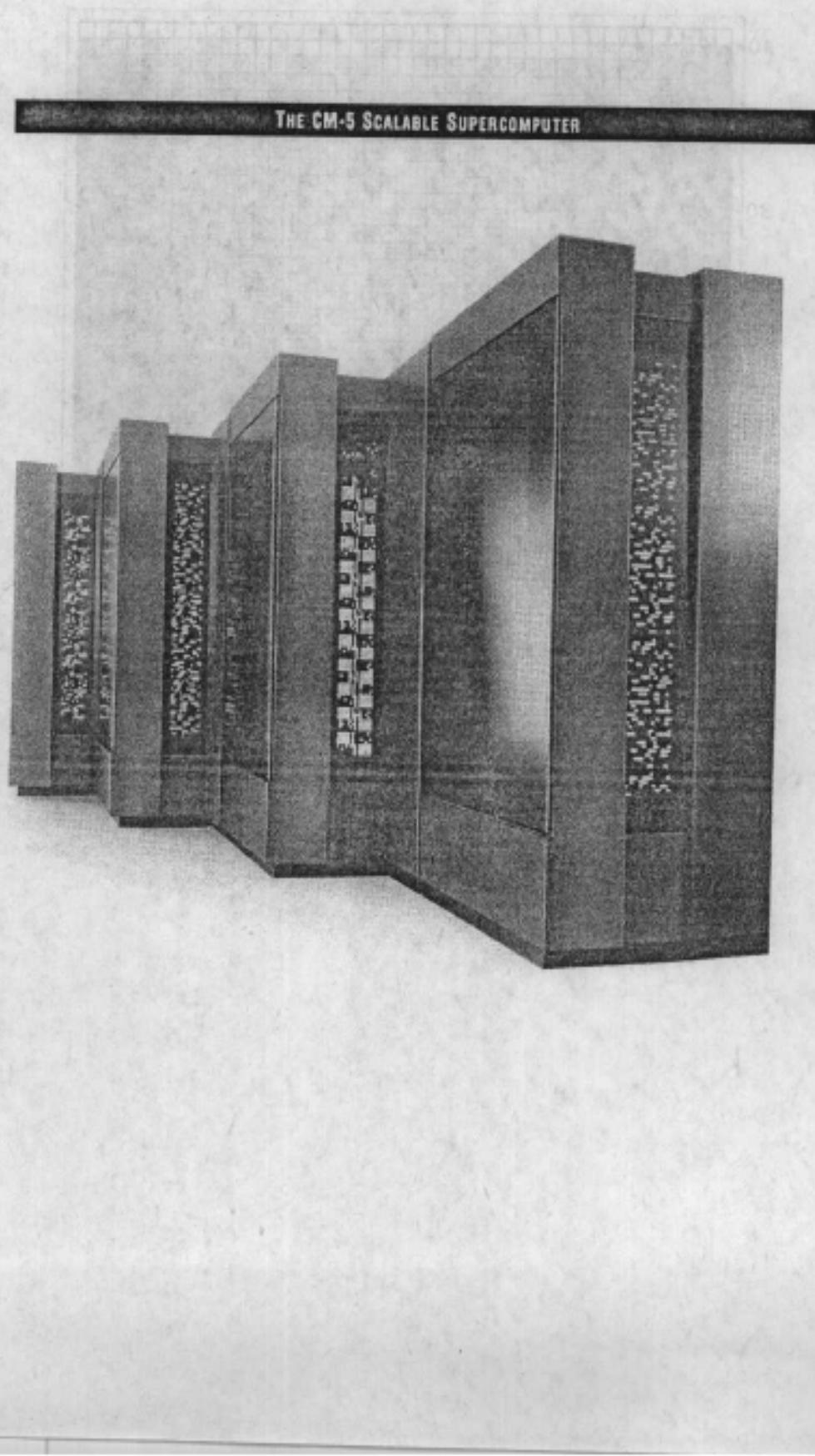


图 10-2 数值结果俯视图

### 附录10.3 安装在国立澳大利亚大学的CM-5系统



## **第四部分 并行计算机编程环境 与分布式程序设计**

**第十一章 PVM**

**第十二章 Linda**

### **概要**

本部分介绍分布式程序设计的重要软件环境 PVM 和 Linda。由于 PVM 具有通用性强及规模小的特点，是目前颇受欢迎的系统，因此，本部分将用大量篇幅介绍 PVM。



# 第十一章 PVM

## § 11.1 PVM 概述

PVM (Parallel Virtual Machine) 是一个能用来进行并行程序设计的软件环境, 它能把一个由异构型计算机构成的网络虚拟成一个大的并行计算机来使用。这些单个的计算机可以是共享主存或分布式主存的多处理器、向量超级计算机, 也可以是专用图形或标量工作站, 甚至 PC 机。它们通过各式各样的网络来进行互连 (如以太网 (ethernet)、FDDI 光纤网等等)。可以说, PVM 几乎能利用目前所有的计算资源去合力解决大型的计算问题。

PVM 最初是由美国 Oak Ridge 国家实验室 (ORNL) 在 1989 年发起研制的, 现在已成为一个由许多大学及研究机构共同参与的大型研究课题。该项目得到了美国能源部、美国国家科学基金会及田纳西州的财政资助。到 1993 年 11 月止, PVM 已发展到 3.2 版。由于 PVM 的源代码是完全公开并且是免费的, 而且它支持的机器类型在不断增加, 再加上其本身的特点, 使得世界上使用 PVM 的人越来越多。

通过 PVM, 用户可以将一些串行、并行或向量式的计算机构成的网络定义成一个具有分布主存的并行计算机来使用。以后, 我们将用虚拟机 (virtual Machine) 这个术语来代替这个逻辑上的具有分布主存的并行机, 而把组成这个虚拟机的各计算机称为宿主机 (host)。PVM 提供在该虚拟机上自动地创建任务以及在任务间进行通讯和实行同步的手段。一个任务被定义成 PVM 中的一个计算单元, 它类似于 UNIX 系统中的进程 (process)。通常, 一个任务可以是一个 UNIX 进程。

PVM 的应用程序可以用 C 语言或 Fortran 语言编写, 它使用信息传递机制来实现并行。信息传递机制对分布主存的计算机都是极常用的。通过发送和接收信息, 应用程序的多个任务可以并行地求解一个问题。

PVM 支持应用程序级、机器级和网络级三个层次上的异构。也就是说, PVM 允许应用程序的任务采用最适合其求解的层次结构。当两台机器使用不同的整型和浮点型数据表示形式时, 可由 PVM 负责所有的数据转换。PVM 甚至允许虚拟机由不同的网络来进行互连。

PVM 系统由两个部分组成, 第一部分是一个监控程序 (daemon), 称为 pvm3 或缩写为 pvm。这个监控程序装在构成虚拟机的每一个宿主机上。pvm3 已设计成可以让每一合法用户都可以在机器上安装这个监控程序。当用户要运行一个 PVM 应用程序时, 他必须先在一台宿主机上运行 pvm3, 则 PVM 将自动地启动所有宿主机上的 pvm3。之后, 用户可以在任一台宿主机的 UNIX 提示符下键入并运行它的 PVM 应用程序。这里的虚拟机是可以由用户自己进行定义的, 并且多个用户所配置的虚拟机间可以相互重叠, 而且每个用户

可以同时运行几个 PVM 应用程序。

PVM 系统的第二部分就是 PVM 接口例程库 (libpvm3.a)，用户程序可以调用这些例程来进行信息传递、创建进程、实现任务同步以及修改虚拟机的配置等。编译 PVM 应用程序时都必须与这个例程库进行连接。

在本书中，我们以 PVM3.0 版为基准来介绍如何使用 PVM 进行并行程序设计。PVM3.0 版较早期版本有较大的改进，这些改进使得 PVM 更易于使用且更加高效。

PVM3.0 主要有以下特点：

- 全新的库函数名以避免与机器上已有库函数名的冲突；
- 采用整数作为任务的标识符，这样使得任务间的通讯更加方便有效；
- 提供强大的进程控制手段，使得用户可以方便地增删宿主机来重新配置虚拟机，灵活地启动和终止 PVM 任务，有效地在 PVM 任务间进行信号传送以及可靠地得到有关虚拟机的配置和活跃的 PVM 任务的信息；
- 提供了容错 (Fault Tolerance) 功能，使得当某台宿主机出现故障时，PVM 能自动地检测到并从虚拟机中删除它，用户还可以通过 PVM 查询宿主机的状态，必要时向虚拟机中添加宿主机以作为替补；
- 实现了动态进程组 (Dynamic Process Group) 的机制，一个进程可以属于多个进程组，而且在计算过程中进程组可以动态地进行切换；
- 提供了两种在 PVM 任务间进行信号传递方法，一种是直接传送一个 UNIX 信号，另一种则是根据用户的标志向一组任务发送信息，由 PVM 负责通知这些任务；
- 提供了在 PVM 任务间进行信息打包和传送的例程。

## § 11.2 启动与配置 PVM

本节将介绍如何用一些机器来配置成一个虚拟机、如何启动 PVM 以及如何使用交互式命令解释程序——PVM 控制台 (console) 或简称 pvm。

PVM 有两种启动方式。一种是先运行控制台程序 pvm，然后通过控制台来增加宿主机；另一种是运行监控程序 pvmd 并根据虚拟机的配置文件 hostfile 来配置虚拟机。如果需要的话，用户可以再启动 PVM 控制台。当不带命令行参数时，pvm 和 pvmd 命令都只在本地宿主机上启动 pvmd。虽然在一个宿主机上用户可以启动多个 PVM 控制台，但通常一个用户只需要启动一个 PVM 控制台。

在这种方式下，用户在 UNIX 提示符下，键入 pvm 命令，则将出现提示符：

pvm>

表明已进入 PVM 交互状态，可以从标准输入设备上接收 PVM 命令。此时，用户可以交互地动态增删宿主机，而且还可以通过它交互地启动和终止 PVM 进程。下面列出 PVM 控制台可接受的命令。

help 或 ? 可用此获取任一 PVM 交互命令的有关信息，以相应的 PVM 命令作为 help 参数，用户将获得有关此命令的选项及标志等详细信息；

version	显示 PVM 接口例程 libpvm 的版本号;
conf	给出虚拟机配置的列表，其中包括宿主机名及 pvm3 的任务号、体系结构类型、最大分块长度和机器的相对速度;
add	把命令后所跟的机器名（一个或多个）加入到虚拟机中;
delete	把命令后所跟的机器名（一个或多个）从虚拟机中删去;
mstat	显示指定宿主机的状态;
ps [-a]	列出当前虚拟机上的所有进程名，以及它们的执行地点、任务号和父任务的任务号;
pstat	显示某一 PVM 进程的状态;
spawn	启动一个 PVM 应用程序;
kill	终止某个 PVM 进程;
reset	终止除 pvm 以外的所有 PVM 进程，消除所有 PVM 内部表项及信息队列，并使监控程序处于空闲状态;
quit	退出 PVM 控制台，但 PVM 监控程序 pvm3 及 PVM 作业继续进行;
halt	终止包括 PVM 控制台在内的所有 PVM 进程，并退出 PVM 系统，所有监控程序也被终止。

在这种方式下，需要用户先准备好一个配置文件 hostfile，用于描述虚拟机的构成及初始配置情况，然后以 hostfile 参数运行 pvm3。用户还可以视需要再启动 PVM 控制台，从而交互地控制 PVM 的运行。

当你在一个终端或单窗口环境中运行 PVM 应用程序后，还想继续做其它事情。这时可以有几种选择。一种是先启动 PVM 控制台，然后在控制台上创建 PVM 作业，再定义虚拟机的配置后退出 PVM 控制台；另一种即是直接在后台启动 PVM，这时，你只需键入：

```
% pvm3 hostfile &
```

需要注意的是，若在 hostfile 中设置了 pw 选项，则这种后台启动方式将无法工作。这是因为 UNIX 不能把用户的输入送给后台作业，所以后台作业将无法收到用户键入的口令。此时可采取下述步骤，先键入：

```
% pvm3 hostfile
```

当 PVM 收到用户键入的口令而完成系统的配置后，将显示 [t80040000] ready。此时再键入 Ctrl-Z+bg，则 PVM 即被放入后台运行。

进入 PVM 的交互状态，在提示符 pvm> 下输入命令 halt，则退出 PVM 系统。

pvm3 命令可以带一个文件名作为参数。在这个文件中列出了用户虚拟机中所包含的宿主机及其相应的初始设置，并且还可以包含希望在以后再放入到虚拟机中的宿主机的有关信息。我们用 hostfile 来表示这个配置文件。用户也可以通过 PVM 控制台程序 pvm 启动 PVM，则它只在本地机上启动 PVM，用户可以通过这个控制台再向虚拟机中添加宿主机。

一个用户安装的 PVM 可以让多个用户使用，但是每个用户都应该有自己的 hostfile 来描述他自己的虚拟机的配置。

最简单的 hostfile 文件格式就是每一行只给出一个机器名，列在第一行的必须是用来初启 PVM 系统的机器。该文件的空白行将被忽略，以#开头的是注解行。

下面的 hostfile 文件定义了一个由 4 台计算机构成的虚拟机（表 11.1，表 11.2）。

表 11.1 一个简单的 hostfile 示例

```
# configuration used for a virtual machines
sparky
azure. epm. ornl. gov
thud. cs. utk. edu
sun4
```

表 11.2 描述所有选项的 PVM hostfile 示例

```
# Comment lines start with # (blank lines ignored)
gstws
ipsc dx=/usr/geist/pvm3/lib/l860/pvmd3
ibml. scri. fsu. edu lo=gst pw

# set default options for following hosts with *
* cp=$sun/problem 1: ~/nla/mathlib
sparky
#azure. epm. ornl. gov
midnight. epm. ornl. gov

# replace default options with new values
* lo=gageist pw ep=problems
thud. cs. utk. edu
speedy. cs. utk. edu

# machines for adding later are specified with &
# these only need listing if options are required
&. sun4 ep=problem1
&. castor dx=/usr/local/bin/pvmd3
&. dasher. cs. utk. edu lo=gageist
&. elvis dx=~/pvm3/lib/SUN $ /pvmd3
```

在机器名所在行的后面还可以跟下列的一些选项，各选项间以空格分隔。

**lo=userid** 允许用户为该宿主机指定一个注册名，否则将使用与初启 PVM 的宿主机相同的注册名。

**pw** PVM 启动时，提示用户输入口令。当用户在远程系统上有不同的注册号和口令时，这一选项特别有用。

**dx=location\_of\_pvmd** 此选项允许用户为该宿主机上的 pvmd 指明一个确定的位置。当某

些用户想使用自己的 pvm3 时，这一选项特别有用。

`ep=paths to user executables` 此选项用于指明一系列路径名，使得当 PVM 在该宿主机上创建任务时，从这些路径上搜索所需要的文件。多条路径名间以冒号隔开。如果不设置该选项，PVM 将在目录 \$HOME/pvm3/bin/ARCH 下寻找。

用户也可以为一组宿主机设置上述某些选项的值。他只需要在某一行以“\*”代替机器名，则出现其后的选项将作为另外一个缺省设置行（以“\*”开头的一行）之前所列出的宿主机的缺省设置。

如果用户不准备把某个宿主机一开始即加入到虚拟机中，而希望在以后再加进来，则可以在相应的行机器名前加一“&.”号。表 11.2 是一个使用了所列选项的 hostfile 文件示例。（在 PVM3.3 版以后，对这些选项稍有改动，请参考相应的手册）。

### § 11.3 编写 PVM 应用程序

PVM 可以支持任何形式的并行结构，即 PVM 的任何一个任务可以在任何时刻与任意其它一个任务并行通讯和同步。两种典型的并行结构是：1. SPMD 模式，其中每个进程大致完成相同的工作；2. master/slave 模式，多个执行计算功能的 slave 进程为一个或多个 master 进程工作。

在 SPMD 模式中，只有一个程序。这种程序有时又叫做 hostless 程序。这仍有一个让所有进程初启的问题。在下列示例中，由用户启动程序的第一个拷贝，然后通过检查 `pvm_parent()`，这个拷贝可以知道自己不是由 `pvm_spawn()` 启动的。如果是第一个拷贝，则启动其它 PVM 进程。这时，所有拷贝都是平等的，它们分别对自己的数据区进行操作并与其它进程协作，采用这种 SPMD 类型的程序不能从 PVM 控制台启动执行，因为若从 PVM 控制台启动，`pvm_parent()` 返回的是控制台任务号，而不是 `PvmNoParent`（一个负值），所以这种类型程序必须在 UNIX 提示符下启动执行。

在 master/slave 模式中，一个 master 程序创建并控制几个执行具体计算的 slave 程序。PVM 并不限于这种模式。例如，任何 PVM 任务都可在别的机器上创建进程，但 master/slave 模式仍然是一个很有意义的模式。master 首先调用 `pvm_mytid()` 使应用程序跟 PVM 建立联系，然后调用 `pvm_spawn()` 用来在宿主机上运行 slave 程序。每个 slave 程序再调用 `pvm_mytid()`，接着调用 `pvm_send()` 和 `pvm_recv()` 在各进程间传递消息。结束时一个 PVM 程序必须调用 `pvm_exit()`，从而让 PVM 关闭进程间通讯的软插座（socket），让该进程脱离与 PVM 的联系。

#### § 11.3.1 C 语言编程示例

##### 1. master/slave 方式

master 源程序：

```
#include "pvm3.h"
#define SLAVENAME "slave"
main()
{
```

```

int mytid;          /* 主任务标识 */
int tids[32];       /* 从任务标识数组 */
int n,nproc,i,who,msgtype;
float data[100],result[32];
/* 向 PVM 登记 */
mytid=pvm_mytid();
/* 输入从任务个数 */
puts("How many slave programs (1-32)?");
scanf("%d",&nproc);
/* 启动从任务 */
pvm_spawn(SLAVENAME,(char * *)0,0,"",nproc,tids);
/* 用户程序开始,初始化数据 */
n=100;
initialize_data(data,n);
/* 将初始数据广播给从任务 */
/* 初始化发送缓冲区 */
pvm_initsend(PvmDataRaw);
/* 将从任务数、从任务标识、数据个数及数据打包 */
pvm_pkint(&nproc,1,1);
pvm_pkint(tids,nproc,1);
pvm_pkint(&n,1,1);
pvm_pkfloat(data,n,1);
pvm_mcast(tids,nproc,0);
/* 等待从任务返回结果 */
msgtype=5;
for (i=0;i<nproc;i++){
    pvm_recv(-1,msgtype);
    pvm_upkint(&who,1,1);
    pvm_upkfloat(&result[who],1,1);
    printf("I got%f from %d\n", result[who],who);
}
/* 程序结束前,退出 PVM */
pvm_exit();
}

```

slave 源程序:

```

#include"pvm3.h"
main()

```

```

{
    int mytid;          /* 任务标识 */
    int tids[32];       /* 从任务标识数组 */
    int n,me,i,nproc,master,msgtype;
    float data[100],result;
    float work();       /* 计算程序 */
    /* 向 PVM 登记 */
    mytid = pvm_mytid();
    /* 接收初始数据 */
    msgtype = 0;
    pvm_recv(-1,msgtype);
    /* 将对应数据解包 */
    pvm_upkint(&nproc,1,1);
    pvm_upkint(tids, nproc,1);
    pvm_upkint(&n,1,1);
    pvm_upkfloat(data,n,1);
    /* 确定任务编号 */
    for (i=0;i<nproc;i++)
        if (mytid == tids[i]) {me = i; break;}
    /* 计算结果 */
    result = work(me,n,data,tids,nproc);
    /* 将计算结果发送给主任务 */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&me,1,1);
    pvm_pkfloat(&result,1,1);
    msgtype = 5;
    master = pvm_parent();
    pvm_send(master, msgtype);
    /* 结束前,退出 PVM */
    pvm_exit();
}

```

## 2. SPMD 方式

```

#define NPROC 4
#include "pvm3.h"
main()
{
    int mytid,tids[NPROC],me,i;
    /* 向 PVM 登记 */
    mytid = pvm_mytid();

```

```

/* 取得父任务名 */
tids[0]= pvm_ parent( );
if(tids[0]<0){
/* 若是第一个任务 */
    tids[0]= mytid;
    me = 0;
/* 创建其余的任务 */
pvm_ spawn("spmd", (char * *)0,0,"",NPROC-1,&tids[1]);
/* 向创建的任务发送信息 */
    pvm_ initSend(PvmDataDefault);
    pvm_ pkint(tids,NPROC,1);
    pvm_ mcast(&tids[1]NPROC-1,0);
}
else {
/* 不是第一个任务,接收广播的信息 */
    pvm_ recv(tids[0],0);
    pvm_ upkint(tids,NPROC,1);
/* 找出本任务的任务编号 */
    for (i=1;i<NPROC;i++)
        if (mytid == tids[i]) {me = i; break;}
}
/* 执行 */
dowork(me,tids,NPROC);
/* 结束前,退出 PVM */
pvm_ exit( );
}

dowork(me,tids,nproc)
int me, * tids, nproc;
{
    int token,dest,count=1,stride=1,msgtag=4;
    if(me == 0){
/* 若是第一个任务 */
        token = tids[0];
/* 初始化发送缓冲区,将任务标识传送给下一个任务 */
        pvm_ initSend(PvmDataDefault);
        pvm_ pkint(&token,count,stride);
        pvm_ send(tids[me+1],msgtag);
/* 等待接收最后一个任务的信息 */
        pvm_ recv(tids[nproc-1],msgtag);
    }
}

```

```

else {
    /* 若不是第一个任务,接收上一个任务的信息 */
    pvm_recv(tids[me-1],msgtag);
    pvm_upkint(&token,count,stride);
    /* 发送给下一个任务 */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&token,count,stride);
    dest = (me == nproc-1)? tids[0]:tids[me+1];
    pvm_send(dest,msgtag);
}
}

```

### § 11.3.2 Fortran 语言编程示例

#### 1. master/slave 方式

master 源程序:

```

program master_1
include "fpvm3.h"
integer i,info,nproc,numt,msgtype,who,mytid,tids(0:32)
double precision result(32),data(100)
character * 12 nodename, arch

c      向 PVM 登记
call pvmfmytid(mytid)
c      输入从任务个数
print *, 'How many slave programs(1-32)? '
read *,nproc
c      启动从任务
nodename = 'slave1'
call pvmfspawn(nodename, PVMDEFAULT, '* ',nproc,tids,numt)
c      用户程序开始,初始化数据
n=100
call initiate..data(data,n)
c      将初始数据广播给从任务
c      初始化发送缓冲区
call pvmfinitsend(0,info)
c      将从任务数、从任务标识、数据个数、数据打包
call pvmfpack(INTEGER4,nproc,1,1,info)
call pvmfpack(INTEGER4,tids,nproc,1,info)
call pvmfpack(INTEGER4,n,1,1,info)

```

```

call pvmfpack(REALS,data,n,1,info)
msgtype = 1
call pvmfmcast(nproc,tids,msgtype,info)
c      等待从任务返回结果
msgtype = 2
do 30 i=1,nproc
    call pvmfreccv(-1,msgtype,info)
    call pvmfunpack(INTEGER4,who,1,1,info)
    call pvmfunpack(REALS,result(who),1,1,info)
30      continue
c      程序结束前,退出 PVM
call pvmfexit()
stop
end

```

slave 源程序:

```

program slave1
include "fpvm3.h"
integer info,mytid,mtid,msgtype,me,tids(0:32)
double precision result,data(100)
double precision work
c      向 PVM 登记
call pvmfmytid(mytid)
call pvmfparent(mtid)
c      接收初始数据
msgtype = 1
call pvmfreccv(mtid,msgtype,info)
c      将对应数据解包
call pvmfunpack(INTEGER4,nproc,1,1,info)
call pvmfunpack(INTEGER4,tids,nproc,i,info)
call pvmfunpack(INTEGER4,n,1,1,info)
call pvmfunpack(REALS,data,n,1,info)
c      确定任务编号
do 5 i=0,nproc
    if(tids(i).eq. mytid)me = i
5      continue
c      计算结果
result = work(me,n,data,tids,nproc)
c      将计算结果发送给主任务

```

```

call pvmfinitSend(PVMDEFAULT,info)
call pvmfpack(INTEGER4,me,1,1,info)
call pvmfpack(REALS,result,1,1,info)
msgtype = 2
call pvmfSend(mtid,msgtype,info)
c      结束前,退出 PVM
call pvmfexit()
stop
end

```

## 2. SPMD 方式

```

program spmd
include "fpvm3.h"
PARAMETER(NPROC=4)
integer mytid,me,numt,i
integer tids(0:NPROC)
c      向 PVM 登记
call pvmfmytid(mtid)
c      取得父任务号
call pvmfparent(tids(0))
if (tids(0).lt.0)then
c      若是第一个任务
tids(0) = mytid
me = 0
c      创建其余的任务
call pvmfspawn('spmd',PVMDEFAULT,'*',NPROC-1,tids(1),numt)
c      向创建的任务发送信息
call pvmfinitSend(0,info)
call pvmfpack(INTEGER4,tids,NPROC,1,info)
call pvmficast(NPROC-1,tids(1),0,info)
else
c      不是第一个任务,接收广播的信息
call pvmfreccv(tids(0),0,info)
call pvmfunpack(INTEGER4,tids,NPROC,1,info)
c      找出本任务的任务编号
do 30 i=1,NPROC-1
    if(mtid.eq.tids(i))me = i
30    continue
    endif
c      执行

```

```

call dowork(me,tids,NPROC)
c      结束前,退出 PVM
call pvmfexit( )
stop
end
subroutine dowork(me,tids,nproc)
integer me,nproc,tids(0:nproc)
integer token,dest,count,stride,msgtag
count = 1
stride = 1
msgtag = 4
if(me.eq.0)then
c      若是第一个任务
token = tids(0)
c      初始化发送缓冲区,将任务标识传递给下一个任务
call pvmfinitsend(0,info)
call pvmfpack(INTEGER4, token, count,stride,info)
call pvmfsend(tids(me+1),msgtag,info)
c      等待接收最后一个任务的信息
call pvmfreccv(tids(nproc-1),msgtag,info)
else
c      若不是第一个任务,接收上一个任务的信息
call pvmfreccv(tids(me-1),msgtag,info)
call pvmfunpack(INTEGER4,token,count,stride,info)
c      发送给下一个任务
call pvmfinitsend(0,info)
call pvmfpack(INTEGER4,token,count,stride,info)
dest = tids(me+1)
c      求下一个任务的编号
if(me.eq.nproc-1)dest = tids(0)
call pvmfsend(dest,msgtag,info)
endif
return
end

```

### § 11.3.3 编写应用程序应该注意的几个问题

应用程序把 PVM 视作一个支持信息传递计算模式的并行计算资源。它具有既通用又灵活的特点。这一资源可以通过三个不同的层次来访问：

(1) 透明模式：在这种模式下，任务能在最合适的宿主机（通常是负载最小的计算

机) 上自动执行。

(2) 依赖于结构的模式: 在这种模式下, 用户可以为运行特定的任务指定特定的体系结构的计算机。

(3) 底层模式: 在这种模式下, 用户可以为运行任务指定特定的计算机。

这样分层方式使得用户在利用网络上单个机器的特殊性能的同时又具有灵活性。

PVM 下的应用程序可以具有任意的控制和从属结构。换句话说, 在一个并发程序的执行过程中, 任何进程之间都可以发生关系。而且任一进程都可以与其它进程进行通信与同步。这一特性使得用户可以实现 MIMD 式的并行计算。但实际上大部分并发应用程序都是很结构化的。两种典型的结构是所有进程都相同的 SPMD 模式和主进程控制运行从进程进行计算的主/从模式。

### 1. 性能上的考虑

PVM 用户可用的程序设计方式有无限种。任何特殊的控制和从属结构都可以在 PVM 系统下通过适当的使用 PVM 结构来实现。另一方面, 应用程序开发者在使用任一信息传递系统进行程序设计时都应当作以下的考虑。

首先要考虑的是任务的粒度。这通常是由一个进程接收的数据量与其执行的浮点运算量的比率来确定的。通过对 PVM 配置中机器的速度和机器间可用网络带宽的一些简单计算, 用户可以得到他所需要的应用程序粒度的下限。粒度的权衡是在大粒度高加速比和大粒度所导致的并行度下降间进行的。

其次要考虑的是信息传递量。接收到的信息可能由很多小的信息组成, 也可能由很少的大信息组成。使用少量的大信息将减少总的信息传递启动时间, 但它不会减少总的执行时间。一些例子表明, 小信息可能与一些计算重叠, 以致它们的开销被屏蔽。通信和计算重叠的能力和最优信息传递数通常由应用程序的特点来决定。

第三个要考虑的是应用更适于功能并行还是数据并行。功能并行是指 PVM 配置中不同的机器执行不同的任务。例如, 一台向量超级计算机可以用来解决问题中适合于向量化部分, 多处理器计算机可以用来解决问题中适合并行的部分, 而图形工作站则用来可视化实时产生的数据。每台机器执行不同的功能(可能使用相同的数据)。

在数据并行模式中, 数据被划分和分布到 PVM 配置中的所有机器上。操作(通常是相似的)在每个数据集上执行, 信息在进程间传递, 直到问题被解决。数据并行在分布存储的多处理机上更为普遍, 因为它只需要写一个在所有机器上执行的并行程序, 并且通常可以扩展到数以百计的处理器上执行。很多线性代表、PDE 和矩阵算法已用数据并行模式开发。

当然, 在 PVM 中两种模式可以混合使用以充分利用每台机器的能力。例如, 在上面功能并行的示例中, 运行于多处理器上的并行代码本身可以是在 PVM 中用数据并行模式来编写的。

### 2. 网络方面的考虑

如果应用程序开发者希望在网络上运行他的并行程序, 他需考虑其它一些因素: 他的并行程序将与其他用户共享网络资源。这种多用户、多任务环境以一种复杂的方式影响其程序的通信和计算性能。

首先要考虑的是在配置中每台机器计算能力不同的影响。因为在虚拟机中按计算能力

的不同，机器被分为不同的集合，不同品牌的工作站具有不同的计算能力。超级计算机间的差异可能更大。甚至用户使用同一类机器时，他也能觉察到每台机器性能上的差异。这是由在配置机的子集上以多任务方式运行多用户的任务所引起的。如果一个用户把他的问题为每台机器分成相同的块（一种并行化的常用方法），那么以上的考虑将产生不利的影响。他的应用程序将和运行在最慢机器上的任务一样慢。如果任务相互协作，最快的机器也将因为等待最慢的任务的数据而变慢。

其次要考虑的是网络的长信息延时所带来的影响。如果正在使用的是广域网，这种影响可能由机器之间的距离造成，也可能由你所在的局域网中你的程序与其他人的程序的竞争所造成。例如 Ethernet 网采用总线结构，那么任一时刻只能有一组信息在上面传送。如果应用程序被设计成每一个任务只给其邻居发送消息，那么可以假设不存在竞争。在一台分布存储的多处理机上没有竞争，所有的发送都可以并行处理。但在 Ethernet 上，发送过程是串行进行的，这导致信息到达相邻的机器有很长的延时。其它网络如令牌环网、FDDI 和 HIPPI，都有可能产生很长的延时，用户必须决定是否将对延时的容忍设计到算法中。

第三个要考虑的是当与其他用户共享这些资源时，计算性能和有效网络带宽是动态变化的。一个应用程序可能在某次运行时有很好的加速比，而在几分钟后的另一次运行时加速比却很差。在一次运行中，一个应用程序可能有正常的同步匹配而不会造成一些任务等待数据。在最坏情况下，一个存在于应用程序中的同步错误只有在特定情况下机器负载发生动态波动时才表现出来。因为这种条件很难重复产生，所以这种类型的错误很难被发现。

在并行应用程序中，许多网络注重于考虑负载平衡的方案。下面将描述一些常用的负载平衡方案。

### 3. 负载平衡

在一个多用户的网络环境中，我们发现负载平衡能较大地提高性能。对于并行程序，有许多实现负载平衡的方案。在这一部分我们将描述三种网络计算中最常用的方案。

最简单的方案是静态的负载平衡。在这种方式下，问题被分块，每一个任务仅分配一次。数据的划分发生在计算开始前，或者在应用程序的开始进行。因为机器计算速度不同，分配给一特定的机器的任务的大小或任务的数量可能要发生变化。因为所有的任务从一开始就是活动的，相互之间可以通信和协调。在一个负载很轻的网络中，静态负载平衡方式可能十分有效。

若计算的负载是变化的，可采用动态负载平衡。最常用的方案被称作任务库变化表。它的典型实现是在主/从模式中，主程序建立并管理一个库，当从程序空闲时发送任务给从程序。库通常实现成一个队列，如果任务的大小发生变化，那么大的任务被放在靠近队列头的地方。在这种方法中，当库中还有任务时从程序一直保持忙状态。这种方法的一个例子在 xep 程序（源程序在 pvm3/mandelbrot 子目录下）中可以看到。因为在这种方法中任务开始和结束的时间是任意的，它很适合于不与其它从程序发生通信，而只与主程序或是文件发生通信的应用程序。

第三种方案不使用主进程，它要求在某个预先规定的时间重新检查和分布各个进程的工作负载。一个例子是求解非线性 PDEs，每一个线性化步骤是静态负载平衡的，在每一个线性步骤进程之间检查问题发生了怎样的变化并重新分布网格点。对于该基本方案可有多种变化。例如不进行所有进程间的同步，而只是将超出的负载分配给它的近邻。或者当进

行负载重分配前一直等待某个进程发出信号，表明它的负载平衡超过了容忍的极限而不是等待一个确定的时间段。

#### § 11.3.4 编译和运行 PVM 应用程序

PVM 应用程序可用 C 语言和 Fortran 语言编写，通过使用 PVM 提供的信息传递机制来实现并行计算。

##### 1. 编译 PVM 应用程序

一个包含 PVM 调用的 C 语言程序必须跟库 libpvm3.a 连接，如果程序用到了动态进程组，那么在跟 libpvm3.a 连接之前还必须跟 libgpvm3.a 连接。如果是 Fortran 程序，则必须与 libfpvm3.a 和 libpvm3.a 连接，如果这个 Fortran 程序也用了动态进程组，则它应该顺序连接 libfpvm3.a、libgpvm3.a 和 libpvm3.a。

在目录 pvm3/example 中除了给出一些 PVM 应用程序实例外还给出一个 makefile 文件，这个 makefile 文件演示了 C 和 Fortran 的应用程序如何连接 PVM 的库，此文件也包含了一些在某些体系结构上要用到的附加库的信息。

##### 2. 运行 PVM 应用程序

如果 PVM 在运行，那么任何调用 PVM 例程的应用程序都可在 UNIX 提示符上直接执行。PVM 应用程序可以在虚拟机的任何宿主机上执行，而不限在初启 PVM 的那台机器。创建的 PVM 子任务的标准输出和标准错误输出写入 PVM 被初启的宿主机上的记载文件/tmp/pvml.<uid> 中。任何交互方式启动的 PVM 任务的标准输出和标准错误都会显示在屏幕上。

在一个 PVM 上可以运行多个应用程序，没有必要为每个应用程序都初启一个新的 PVM，只有当一个应用程序运行混乱时才应该重新启动 PVM。

需要注意的是，在非正常退出 PVM 之后，再次启动 PVM 之前，要删除掉 PVM 的两个内部文件/tmp/pvml.<uid> 和 /tmp/pvml.<uid>，这里的 uid 表示用户的注册名。

#### § 11.3.5 程序调试

一般来说，调试并行程序比调试串行程序复杂，不仅因为有多个进程同时在运行，并且它们间的相互作用也会产生错误。比如，一个处理器接收错误的数据而导致以后发生除 0 错。另外一个例子是死锁，当一个程序错误导致所有的进程等待信息时就会造成死锁。如果执行时检测到错误，所有的 PVM 子程序都将返回一个错误码。下表中给出这些代码和它们的含义。

PVM 提供的两个子程序 pvm\_perror() 和 pvm\_serror() 允许在 PVM 子程序中自动检测和打印任何错误码。要记住 PVM 任务可以从任何一个标准的串行调试器下启动，比如 dbx。

被产生的 PVM 任务也可以在一个调试器下产生。通过在调用 pvm\_spawn() 时设置包含 PvmTaskDebug 的 flag 参数，PVM 将执行外壳描述 pvm3/lib/debugger。在实现这一描述时，在宿主机上启动一个 xterm 窗口来运行 PVM，在调试器下创建的任务也在这个窗口中运行。被调试的任务可以在虚拟机上的任何一个节点机上执行，这可以通过指定 pvm\_spawn() 中的 flag 和 where 参数来实现。这个外壳描述可以修改成包含一个更可取的调

试器，如果说有的话甚至可以是一个并行调试器。

创建任务的诊断打印语句的结果发送给了 stderr 或 stdout 而不会在用户屏幕上显示出来。所有这些都被定向到启动 PVM 的宿主机上的文件/temp/pvml. (uid) 中。

经验告诉我们，调试一个 PVM 程序分以下三步：

首先，如果可能，用一个进程运行这一程序，并像调试一个串行程序那样来调试。这一步的目的是找出与并行化无关的下标和逻辑错误。一旦这些错误被纠正，就进行下一步。

其次，在一台计算机上用 2 到 4 个进程来运行这一程序。在 hostfile 中仅放入一台工作站的名字。PVM 将在一台机器上运行这些进程。这一步的目的是检查通信语法和逻辑。比如，一个用等于 4 的 tag 发送的信息用等于 5 的 tag 接收。一种在这一步中发现的更常见的错误是使用了不唯一的信息。作为一个例子，假定同样的信息记号总是被使用，那么一个进程从三个分散的信息中接收一些初始化数据，但是它没有办法决定一个信息代表哪一个数据。PVM 返回任何一个满足所要求的发送者和信息记号的数据，所以应由用户来确定联系一个信息的唯一的标识。一个不唯一的信息记号错是很难查出的，因为这种错误对微妙的同步性是很敏感的，并且是难以再现的。如果这种错误不能通过 PVM 返回的错误码和一条打印语句来确定，那么用户可以通过在调试器下启动一个或所有的任务来获得对程序的完全的调试控制权。这就允许对每一个在调试工具下运行的进程设置断点、变量跟踪、单步跟踪以及向后跟踪，甚至在它同其它 PVM 任务传送信息时也可如此。

第三步是在几台机器上运行这 2 到 4 个进程。这一步的目的是检查由网络延时所产生的同步错误。在这一步中发现的错误敏感于算法中信息到达的次序，程序死锁通常由敏感于网络延时的逻辑错误所造成。同第二步一样可以进行完全控制的调试，但这样做的效果不大，因为调试器会转换或隐藏前面所提到的与时间有关的错误。表 11.3 列出有关的错误码。

表 11.3 PVM 返回的错误码

错误名	错误码	意义
PvmOk	0	正常
PvmBadParam	-2	错误的参数
PvmMismatch	-3	屏障数不匹配
PvmNoData	-5	读到了缓冲区尾
PvmNoHost	-6	没有这样的宿主机
PvmNoFile	-7	没有这样的执行文件
PvmNoMem	-10	没有足够的内存
PvmBadMsg	-12	不能对接收到的信息解码
PvmSysErr	-14	pvmrd 没有反应

续表

错误名	错误码	意义
PvmNoBuf	-15	没有当前缓冲区
PvmNoSuchBuf	-16	错误的信息代号
PvmNullGroup	-17	空的进程组名非法
PvmDupGroup	-18	进程已经在进程组中
PvmNoGroup	-19	没有那样的进程组名
PvmNotInGrou	-20	不在进程组中
PvmNoInst	-21	进程组中没有这样的实例
PvmHostFail	-22	宿主机错误
PvmNoParent	-23	没有父任务
PvmNotImpl	-24	函数没有实现
PvmDSysErr	-25	pvmrd 系统错误
PvmBadVersion	-26	pvmrd 与 pvmrd 版本不一致
PvmOutOfRes	-27	资源用完了
PvmDupHost	-28	宿主机已经在虚拟机中
PvmCantStart	-29	执行新的 pvmrd 失败
PvmAlready	-30	从 pvmrd 已经运行
PvmNoTask	-31	任务不存在

## § 11.4 PVM 库函数使用指南

### § 11.4.1 进程控制类函数

1. `pvm_mytid()`, `pvmfmytid()`

调用格式:

```
int tid = pvm_mytid (void)
call pvmfmytid (tid)
```

功能: 在第一次调用时, 将调用进程注册为 PVM 进程。在每一次调用时, 都返回调用进程的 tid。

参数:

`tid`: 返回调用进程的 tid, 负值表示出错。

描述:

在任何调用 PVM3.0 子程序的程序中, pvm\_mytid() 必须是第一个调用的 PVM 子程序。在这个子程序的第一次调用时, 它将调用进程注册为 PVM 进程, 如果调用进程不是由 pvm\_spawn() 创建的, 它给调用进程赋予一个唯一的 tid。在一个应用软件中, 该子程序可调用多次, 每次调用时均返回调用进程的 tid。

tid 是由本地 pvmfd 赋予的一个 32 位正整数, 这个整数的 32 位分为几个域, 分别存放调用进程的不同信息, 如它在虚拟机中的位置(即本地 pvmfd 地址), 当调用进程是在多处理机上时的 CPU 数、进程 ID。这些信息是供 PVM 系统使用的, 用户不要去修改它。

如果应用软件调用 pvm\_mytid() 以前没有启动 PVM 的话, 则 tid 返回负值。

**错误信息:**

pvm\_mytid 返回如下错误码:

错误名	可能的原因
PvmSysErr	pvmfd 不响应

### 2. pvm\_exit(), pvmfexit()

**调用格式:**

```
int info=pvm_exit(void)
call pvmfexit(info)
```

**功能:** 通知本地 pvmfd, 调用进程要退出 PVM。

**参数:**

info: 返回的整数状态码, 负值表示出错。

**描述:**

通知本地 pvmfd, 调用进程要退出 PVM。该子程序并不终止调用进程, 仅仅是使调用进程脱离与 PVM 的联系, 此后, 调用进程与普通的 UNIX 进程一样, 继续进行它的工作。

所有的 PVM 进程在终止前最好都调用 pvm\_exit。对于不是用 pvm\_spawn() 启动的 PVM 进程, 在其终止前必须调用 pvm\_exit, 否则, PVM 不知道该进程是否正常终止了。

**错误信息:**

pvm\_exit 返回如下错误码:

错误名	可能的原因
PvmSysErr	pvmfd 不响应

### 3. pvm\_spawn(), pvmfspawn()

**调用格式:**

```
int numt = pvm_spawn(char * task,char ** argv,int flag,char * where,int ntask,
                     int * tids)
call pvmfspawn(task, flag, where, ntask, tids, numt)
```

**功能:** 创建一个新的 PVM 进程。

**参数:**

task: 一个字符串, 存放要创建 PVM 进程所执行的可执行文件名。该可执行文件应在

进程要创建的计算机上存在。PVM 查找该可执行文件的缺省位置为 \$HOME/pvm3/bin/ARCH。

**argv:** 一个指向数组的指针, 该数组中存放了可执行文件所用的参数。为 NULL 的数组元素表示参数的结束。若执行文件不带参数, 第二个参数为 NULL。

**flag:** 创建 PVM 进程时所用的选项。

在 C 语言中可取下列值:

PvmTaskDefault	0	PVM 任选一台计算机, 在其上创建任务。
PvmTaskHost	1	where 参数指出在其上创建任务的特定计算机
PvmTaskArch	2	where 参数指出在其上创建任务的特定机器型。
PvmTaskDebug	4	创建任务在调试器的控制下。
PvmTaskTrace	8	创建任务将产生 PVM 的轨迹数据 (今后扩充)。

在 FORTRAN 语言中可取下列值:

PVMDEFAULT	0	PVM 任选一台计算机, 在其上创建任务。
PVMHOST	1	where 参数指出在其上创建任务的特定计算机
PVMARCH	2	where 参数指出在其上创建任务的特定机器型。
PVMDEBUG	4	创建任务在调试器的控制下。
PVMTRACE	8	创建任务将产生 PVM 的轨迹数据 (今后扩充)。

**where:** 一个字符串, 指定在什么地方创建 PVM 进程, 根据 flag 值的不同, where 可以是机器名, 也可以是机器类型名。若 flag=0, 就忽略 where 参数, PVM 自己选择一个最合适的选择机。

**ntask:** 要创建的执行同一个可执行文件的 PVM 进程数。

**tids:** 一个具有 ntask 个元素的数组, 返回时, 每个数组元素中存放一个由该调用创建的 PVM 进程的 tid。若创建某个任务时出错, 则在其对应元素中存放相应的错误码。

**numt:** 整数返回时存放实际创建的任务数。负值表示系统出错。小于 ntask 的正数表示创建部分失败。在这种情况下, 用户应检查 tids 数组中的错误代码。

**描述:**

pvm\_spawn() 创建 ntask 个 PVM 进程, 它们均执行由 task 指定的可执行文件。新创建的 PVM 进程所在的计算机由 flag 和 where 参数指定。返回时, 数组 tids 中包含每一个新创建任务的 tid, 若创建一个或多个任务的话, numt 返回实际创建的任务数。若出现系统错误的话, 返回负值。若小于 ntask, 则有一些任务创建未能成功。用户应检查 tids 数组的最后几个元素中的错误码, 其含义见错误信息。

当 flag 置为 0, where 置为 NULL (在 FORTRAN 中为 \*) 时 PVM 使用启发式方法来将 ntask 个任务分布到虚拟机上, 最初, 启发式方法将用机器负载的尺度和性能比率来确定最合适的选择机。

当 where 指定的是一个多处理器的特殊情况时, 将使用厂家提供的子程序在这个机器上创建所有的 ntask 个任务。

当设置为 PvmTaskDebug 时, pvm\_spawn 将在调试器中创建任务, 在这种情况下, 不是执行 pvm3/bin/ARCH/task args, 而是执行 pvm3/lib/debugger pvm3/bin/ARCH/

task args, 这里 pvm3/lib/debugger 是一个 shell 程序, 用户可以修改它, 进行自己的尝试。目前, 这个 shell 程序启动一个 X 窗口, 在其中有与 dbx 兼容的调试器。

错误信息:

pvm\_spawn() 返回的错误状态如下:

PvmBadParam	-2	参数值无效
PvmNoHost	-6	指定的计算机不属于虚拟机
PvmNoFile	-7	找不到指定的可执行文件, 记住 PVM 查找可执行文件的缺省路径为 \$HOME/pvm3/bin/ARCH。这里, ARCH 是该计算机的结构类型名
PvmNoMem	-10	内存分配失败, 即在计算机上没有足够的内存
PvmSysErr	-14	没有响应
PvmOutOfRes	-27	资源消耗完了

#### 4. pvm\_kill(), pvmfkill()

调用格式:

```
int info = pvm_kill (int tid)
call pvmfkill (tid, info)
```

功能: 终止指定的进程

参数:

tid: 要被终止的 PVM 进程的 tid。

info: 一个整数, 存放该子程序返回的状态码, 负值表示出错。

描述:

pvm\_kill() 发送一个终止信号给由 tid 指定的 PVM 进程, 若调用成功, info 返回 0, 否则返回负值。

pvm\_kill() 不能用来终止调用进程。在 C 语言中, 要终止本进程应调用 pvm\_exit(), 然后跟 exit(); 在 FORTRAN 语言中, 要终止本进程应调用 pvmfexit(), 然后跟一个 stop 语句。

错误信息:

pvm\_exit 返回如下错误码:

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法

#### § 11.4.2 信息类函数

##### 1. pvm\_parent(), pvmfparent()

调用格式:

```
int tid = pvm_parent (void)
call pvmfparent (tid)
```

功能: 返回调用进程的父进程 tid。

参数:

**tid:** 返回的调用进程父进程的 tid。如果调用进程不是用 pvm\_spawn() 创建的，返回 PvmNoParent。

**描述:**

在 SPMD 方式的程序中，一个给定的程序实例可以调用它来确定是否需要创建一个跟自己一样的 PVM 进程，前面有例子说明这种用法。在主从方式的程序中，从任务可以调用它来确定主任务的 tid，以便将结果回送给主任务。

**错误信息:**

pvm\_parent 返回如下错误码：

错误名	可能的原因
PvmNoParent	调用进程不是由 pvm_spawn() 创建的。

### 2. pvm\_mstat(), pvmfmstat()

**调用格式:**

```
int mstat = pvm_mstat (char * host)
call pvmfmstat (host, mstat)
```

**功能:** 返回虚拟机中指定计算机的状态。

**参数:**

**host:** 存放计算机名的字符串。

**mstat:** 返回的计算机状态，为整数，其取值及含义如下：

PvmOk	计算机正常
PvmNoHost	指定计算机不在虚拟机中
PvmHostFail	指定的计算机不可用

**描述:**

pvm\_mstat() 返回由 host 指定的计算机的状态，可以调用该子程序来确定一个指定的计算机是否出了故障，虚拟机是否需要重构。

**错误信息:**

pvm\_mstat 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmNoHost	所给宿主机不在虚拟机中
PvmHostFail	所给宿主机不可用

### 3. pvm\_pstat(int tid), pvmfpstat()

**调用格式:**

```
int status = pvm_pstat (tid)
call pvmfpstat (tid, status)
```

**功能:** 返回指定的 PVM 进程的状态

**参数:**

**tid:** 要查询状态的 PVM 进程的 tid

**status:** 返回有 tid 指定的 PVM 进程的状态。如果指定任务正在运行，返回状态为 PvmOk；若没有运行，返回状态为 PvmNoTask；若 tid 无效，返回状态为 Pvm-

BadParam。

描述：

pvm\_pstat( ) 返回由 tid 指定的进程的状态。

错误信息：

pvm\_pstat 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法，很有可能 tid 值不合法
PvmNoTask	任务 tid 不在运行

#### 4. pvm\_config( ), pvmfconfig( )

调用格式：

```
int info = pvm_config (int *nhost, int *narch, struct hostinfo **hostp)
struct hostinfo {
    int hi_tid;
    char *hi_name;
    char *hi_arch;
    int hi_mtu;
    int hi_speed;
} hostp;
call pvmfconfig (nhost, narch, dtid, name, arch, speed, info)
```

功能：返回当前虚拟机配置的有关信息。

参数：

C:

nhost：一个整型量，返回时存放虚拟机中的计算机数目。

narch：一个整型量，返回时存放虚拟机中所用到的不同数据格式的数目。

hostp：一个指向结构数组的指针。该数组的每一个元素各存放一台计算机的有关信息。这些信息是 spmd 的 tid，该计算机的名字，该计算机的结构类型，最大包长度、相对运行程度。

info：一个整数，存放该子程序返回的状态码，负值表示出错。

FORTRAN:

nhost：一个整型量，返回时存放虚拟机中的计算机数目。

narch：一个整型量，返回时存放所用到的不同数据格式的数目。

dtid：pvmrd 的 tid。

name：该计算机的名字。

arch：该计算机的结构类型。

speed：该计算机的相对运行程度。

info：一个整数，存放该子程序返回的状态码，负值表示出错。

描述：

返回当前虚拟机的有关信息，这些信息与控制台的 conf 命令所得到的信息相似。

若调用成功，info 返回 0，否则返回负值。

错误信息：

pvm\_config 返回如下错误码：

错误名	可能的原因。
-----	--------

PvmSysErr	pvmrd 不响应。
-----------	------------

5. pvm\_tasks ( ), pvmftasks ( )

调用格式：

```
int info = pvm_tasks (int where, int * ntask, struct taskinfo ** taskp)
struct taskinfo {
    int ti_tid;
    int ti_ptid;
    int ti_host;
    int ti_flag;
    char * ti_a, out;
}taskp;
call pvmftasks (where, ntask, tid, ptid, dtid, flag, aout, info)
```

功能：返回运行在虚拟机上任务的有关信息。

参数：

C:

where：指出要对哪些任务返回有关信息，其可能的取值与含义如下：

0 在虚拟机上的所有任务。

pvmrd tid 指定的 pvmrd 所在计算机上的所有任务。

tid 指定任务。

ntask：整数，返回时存放报告了有关信息的任务个数。

taskp：一个指出结构数组的指针，其数组的每一个元素为一个结构，存放一个任务的有关信息，如 tid，父任务 tid，其 pvmrd 的 tid，状态标志，该任务所执行的可执行文件名。状态标志的取值有：等待消息，等待 pvmrd，正在运行。

info：一个整数，存放该子程序返回的状态码，负值表示出错。

FORTRAN:

where：指出要对哪些任务返回有关信息，其可能的取值与含义如下：

0 在虚拟机上的所有任务。

pvmrd tid 指定的 pvmrd 所在计算机上的所有任务。

tid 指定任务。

ntask：整数，返回时存放报告了有关信息的任务个数。

tid：该任务的 tid。

ptid：父任务的 tid。

dtid：其 pvmrd 的 tid。

flag：状态标志。

aout: 该任务所执行的可执行文件名。

info: 一个整数, 存放该子程序返回的状态码, 负值表示出错。

描述:

pvm\_tasks() 返回当前运行在虚拟机上任务的有关信息。这些信息与控制台上 ps 命令所得到的信息相同。

错误信息:

pvm\_tasks 返回如下错误码:

错误名	可能的原因,
PvmSysErr	pvmrd 不响应。
PvmBadParam	参数不合法, 很有可能 tid 值不合法。
PvmNoHost	所给宿主机不在虚拟机中。

### § 11.4.3 动态配置类函数

#### 1. pvm\_addhost(), pvmfaddhost()

调用格式:

```
int info = pvm_addhost (char **hosts, int host, int *infos)
call pvmfaddhost (host, info)
```

功能: 在虚拟机中增加一台或多台计算机。

参数:

hosts: 一个具有 nhost 个元素的指向字符串的指针数组, 每一个指针所指的字符串中存放一个要增加的计算机的机器名, 调用该子程序时, 要增加的计算机上必须已安装了 pvmrd, 且用户在要增加的计算机上有账号。

nhost: 一个整数, 指定要增加的计算机数目。

infos: 一个具有 nhost 个元素的整型数组, 存放该子程序为相应计算机返回的状态码, 负值表示出错。

host: 一个包含要增加计算机名的字符串。

info: 一个整数, 存放该子程序返回的状态码, 负值表示出错。

描述:

pvm\_addhost() 将一组由 hosts 指定的计算机增加到虚拟机中, 如果调用成功, 则 info 返回 0, 否则返回负值。当 info 返回负值时, 用户可以检查数组 infos 来查看是增加哪一台计算机时出的错。

FORTRAN 子程序 pvmfaddhost() 一次只能在虚拟机中增加一台计算机。如果一台计算机出了故障, PVM 将继续运行, 用户可以使用本子程序来增加应用软件的容错性能, 即应用软件可以使用 pvm\_mstat() 和 pvm\_config() 来得到计算机的状态。如果一台计算机出了故障, PVM 则自动地将该计算机从虚拟机中删去。应用软件可以调用 pvm\_addhost() 来增加一台替代的计算机。

使应用软件具有对计算机故障的容错能力是应用软件开发者的责任。该子程序的另一个用途就是当某些时候有更多的计算机空闲时, 如周末时, 可以将它们加到虚拟机中去。如果应用软件能动态地检测虚拟机配置的话, 它就可以利用到更多的计算能力。

pvm\_addhost( ) 和 pvm\_delhost( ) 均是高开销的操作，要求整个虚拟机同步。

错误信息：

pvm\_addhost 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法
PvmOutOfRes	PVM 用尽了系统资源

#### 2. pvm\_delhost( ), pvmfdelhost( )

调用格式：

```
int info = pvm_delhost (char ** hosts, int host, int * infos)
call pvmfdelhost (host, info)
```

功能：在虚拟机中删除一台或多台计算机。

参数：

host：一个具有 nhost 个元素的指向字符串的指针数组，每一个指针所指的字符串中存放一个要删除的计算机的机器名。

nhost：一个整数，指定要删除的计算机数目。

infos：一个具有 nhost 个元素的整型数组，存放该子程序为相应计算机返回的状态码，负值表示出错。

host：一个包含要删除计算机机器名的字符串。

info：一个整数、存放该子程序返回的状态码，负值表示出错。

描述：

pvm\_delhost( ) 将一组由 hosts 指定的计算机从虚拟机中删除。运行在这些机器上的所有 PVM 进程和 pvmrd 都将在计算机被删除前终止，如果调用成功，则 info 返回 0，否则返回负值。当 info 返回负值时，用户可以检查数组 infos 来查看是删除哪一台计算机时出的错。

FORTRAN 子程序 pvmfdelhost( ) 一次只能在虚拟机中删除一台计算机。

错误信息：

pvm\_delhost 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法
PvmOutOfRes	PVM 用尽了系统资源

#### § 11.4.4 信号函数

##### 1. pvm\_sendsig( ), pvmfsendsig( )

调用格式：

```
int info=pvm_sendsig (int tid, int signum)
call pvmfsendsig (tid, signum, info)
```

功能：给另一个 PVM 进程发信号。

**参数：**

- tid：接收信号的 PVM 进程的 tid。
- signum：信号号码，整数。
- info：一个整数，存放该函数返回的状态码，负值表示出错。

**描述：**

pvm\_sendsig（）给由 tid 指定的进程发指定的信号。若 pvm\_sendsig（）调用成功，info 返回 0，否则返回负值。

pvm\_sendsig（）是提供给具有信号处理经验的程序员使用的，在一个并行环境，它很容易引入不确定性、死锁甚至系统崩溃。建议用户尽量避免使用。

**错误信息：**

pvm\_sendsig 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmd 不响应
PvmBadParam	参数不合法

## 2. pvm\_notify（），pvmfnotify（）

**调用格式：**

```
int info = pvm_notify (int what, int msgtag, int ntask, int * tids)
call pvmfnotify (what, msgtag, ntask, tids, info)
```

**功能：**对一组任务启动某个事件的事件通知机制。所谓事件通知机制，就是当某个事件发生时，PVM 能自动地向一组任务发送信息，通知该事件发生。

**参数：**

what：一个整数，表示对什么事件启动事件通知机制。当前可能的取值如下：

PvmTaskExit	1	任务终止
PvmHostDelete	2	删除计算机
PvmHostAdd	3	增加计算机

msgtag：用于通知事件发生时所用的信息类型。

ntask：要通知的任务个数。

tids：一个具有 ntask 个元素的整型数组，其每个元素存放一个要通知的任务的 tid。

info：一个整数，存放该例程返回的状态码，负值表示出错。

**描述：**

pvm\_notify（）是一个启动事件通知机制的子程序。在调用它以后，如果指定事件发生，PVM 将自动地产生一个信息并发送给由 tids 数组指定的一组任务，这组任务应用 msgtag 信息类型来接收这一信息，并根据接收到的消息采取适当的行动。目前尚不支持用户自定义事件。

**错误信息：**

pvm\_notify 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmd 不响应
PvmBadParam	参数不合法

### § 11.4.5 错误信息处理函数

1. pvm\_perror ( ), pvmferror ( )

调用格式:

```
int info = pvm_perror (char *msg)
call pvmferror (msg, info)
```

功能: 显示最后一个 PVM 调用的错误信息。

参数:

msg: 用户提供的一个字符串, 它将加在最后一个 PVM 调用的错误信息后显示。

描述:

pvm\_perror ( ) 返回最后一个 PVM 调用的错误信息。用户可以利用 msg 在错误信息后显示一些附加的信息, 比如调用任务的位置。所有的标准输出和标准错误输出信息都存放在主 pvmd 所在计算机的文件/tmp/pvmd.〈uid〉中。

错误信息:

本调用不返回任何错误信息。

2. pvm\_serror ( ), pvmferror ( )

调用格式:

```
int oldset = pvm_serror (int set)
call pvmferror (set, oldset)
```

功能: 为以后的 PVM 调用打开或关闭自动错误信息输出机制。

参数:

set: 决定是否打开自动错误信息输出机制。1 为打开, 0 为关闭。

oldset: 返回以前的自动错误信息输出设置。

描述:

pvm\_serror ( ) 为该进程以后的 PVM 调用设置自动错误信息输出方式。当打开时 (为 1), 任何返回错误条件的 PVM 子程序将自动地输出有关错误信息。关闭时 (为 0) 不自动输出有关错误信息。以前的自动错误输出设置返回在 oldset 中。

错误信息:

pvm\_serror 返回如下错误码:

错误名	可能的原因
PvmBadParam	参数不合法

### § 11.4.6 信息传递类函数

在 PVM 中发送信息分三步: 首先调用 pvm\_initsend ( ) 或 pvm\_mkbuf ( ) 初始化发送缓冲区, 第二步, 调用多个 pvm\_pk \* ( ) 的组合将信息填充进缓冲区 (在 FORTRAN 中所有信息填充都是用 pvmfpack ( ) 函数实现的), 第三步调用 pvm\_send ( ) 函数将信息发送给另一进程, 或调用 pvm\_mcast ( ) 函数将信息广播给多个进程。

接受信息者可调用阻塞接收子程序 pvm\_recv ( ) 或非阻塞接收子程序 pvm\_nrecv ( ), 然后调用 pvm\_upk \* ( ) 子程序从接收缓冲区中卸出信息内容。接收函数可被设置为接收

任何信息，或接收从某一点发来的任何信息，或接收带有特定标志的信息。

PVM 还提供了一个探测函数 pvm\_prob() 用来检查一个信息是否已经到达。

PVM3.1 版中新增加了一个 pvm\_advise() 函数。该函数“建议”在后续的信息传递中是否采用任务间直接连接来支持通信。可多次调用 pvm\_advise()，但最典型的情况就是在调用 pvm\_mytid() 后跟着调用它。利用任务间直接连接可提高性能，但由于 UNIX 系统只允许少量这种直接连接，所以它的使用受到限制。

### 1. 信息缓冲区函数

当用户在一个应用软件中要管理多个信息缓冲区时，就需要用到下列函数，多数进程之间传递信息是用不着多个信息缓冲区的，PVM2.4 中只有一个发送缓冲区和一个接收缓冲区。在 PVM3 的实现中，任何时候只有一个活跃的发送缓冲区和一个活跃的接收缓冲区，但程序员可创建多个缓冲区，并轮流把它们切换为活跃的，所有填充、发送、接收卸出函数都只对活跃缓冲区起作用。

(1) pvm\_mkbuf(), pvmfmkbuf()

调用格式：

```
int bufid = pvm_mkbuf (int encoding)
call pvmfmkbuf (encoding, bufid)
```

功能：创建一个新的信息发送缓冲区。

参数：

encoding：一个整数，用来指定信息的数据格式转换方法。

在 C 语言中可能的使用值如下：

PvmDataDefault	0	如虚拟机由同构型机器构成，则不进行数据格式转换，否则采用 XDR（外部数据表示）来进行数据格式转换
----------------	---	---

PvmDataRaw	1	不进行数据格式转换
------------	---	-----------

PvmDataInPlace	2	数据仍存放在原处
----------------	---	----------

在 Fortran 语言中可能的使用值如下：

PVMDEFAULT	0	如虚拟机由同构型机器构成，则不进行数据格式转换，否则采用 XDR（外部数据表示）来进行数据格式转换
------------	---	---

PVMRAW	1	不进行数据格式转换
--------	---	-----------

PVMINPLACE	2	数据仍存放在原处
------------	---	----------

bufid：一个整型量，返回时存放信息缓冲区的标识，负值表示出错。

描述：

pvm\_mkbuf() 创建一个新的信息缓冲区并指定组装信息时所采用的数据格式转换方法。若 pvm\_mkbuf() 调用成功，bufid 中返回新缓冲区标识，这个缓冲区以后可以用做发送缓冲区。若调用失败，bufid 返回负值。当用户打算管理多信息缓冲区时，就可以调用 pvm\_mkbuf() 来创建一个信息发送缓冲区，在信息发送以后并且不再需要发送缓冲区时，应调用 pvm\_freebuf() 释放发送缓冲区。接收缓冲区由 pvm\_recv() 和 pvm\_nrecv() 自动建立，除非你用 pvm\_setrbuf() 显式保存过，否则不必释放。在大多数情况下用户并不需要使用多个发送和接收缓冲区，用户只需要简单地

用 pvm\_initsned ( ) 来清除发送缓冲区即可。

当使用多缓冲区时，通常是对每一个被组装的信息先创建一个缓冲区，然后再释放。

错误信息：

pvm\_mkbuf 返回如下错误码：

错误名	可能的原因
PvmBadParam	参数不合法
PvmNoMem	没有足够内存创建该缓冲区

## (2) pvm\_initsend ( ), pvmfinitSend ( )

调用格式：

```
int bufid = pvm_initsend (int encoding)
call pvmfinitSend (encoding, bufid)
```

功能：清缺省的发送缓冲区并指定信息的数据格式转换方法。

参数：

encoding：一个整数，用来指定信息的数据格式转换方法。

在 C 语言中可能的使用值如下：

PvmDataDefault	0	如虚拟机由同构型机器构成，则不进行数据格式转换，否则采用 XDR（外部数据表示）来进行数据格式转换
PvmDataRaw	1	不进行数据格式转换
PvmDataInPlace	2	数据仍存放在原处

在 Fortran 语言中可能的使用值如下：

PVMDEFAULT	0	如虚拟机由同构型机器构成，则不进行数据格式转换，否则采用 XDR（外部数据表示）来进行数据格式转换
PVMRAW	1	不进行数据格式转换
PVMINPLACE	2	数据仍存放在原处

bufid：一个整型量，返回时存放信息缓冲区的标识，负值表示出错。

描述：

pvm\_initsend ( ) 重清发送缓冲区以便为组装新的信息做准备。组装时所采用的数据格式转换方法由 encoding 指定，缺省时，若虚拟机中的所有计算机有相同的数据格式，则不进行数据格式转换，否则采用 XDR 来完成数据格式转换。其它的选项使得用户即使在虚拟机是异构型时也可以利用他对虚拟机的知识，例如，如果用户知道下一个信息仅发送给那些可以解释其原始数据格式的机器的话，他可以指定 PvmDataRaw 数据格式转换方法来避免数据格式转换的开销。

PvmDataInPlace 使得信息组装时数据项仍留在原处。信息缓冲区中仅包含要发送数据项大小和指向它存储位置的指针，当调用 pvm\_sned ( ) 时，这些数据项直接从用户存储区中取出。该选项避免了复制数据项到信息缓冲区中的开销，但用户必须保证在组装数据项以后到发送它们以前不去修改要发送的数据项。

在最初的 PVM3.0 中，缺省项时总是采用 XDR 的数据格式转换，因为 PVM 并不知道用户是否在信息发送前将增加某个异构的机器，此外，在最初的版本中，PvmDataInPlace 也未实现。若 pvm\_initsend() 调用成功，bufid 中返回发送缓冲区标识，否则返回负值。参见 pvm\_mkbuf()。

错误信息：

pvm\_initsend 返回如下错误码：

错误名	可能的原因
PvmBadParam	参数不合法
PvmNoMem	没有足够内存创建该缓冲区
(3) pvm_freebuf(), pvmffreebuf()	

调用格式：

```
int info = pvm_freebuf (int bufid)
call pvmffreebuf (bufid, info)
```

功能：释放一个信息缓冲区。

参数：

bufid：信息缓冲区标识号  
info：例程返回的整数状态码，负值表示出错。

描述：

释放由 bufid 指定的信息缓冲区，当信息发送以后，并且不再需要该缓冲区时，应调用 pvm\_freebuf() 释放该缓冲区。接收缓冲区通常不必释放，除非你在使用多缓冲区过程中保存过接收缓冲区，但是 pvm\_freebuf() 也可以用来破坏接受缓冲区。在应用软件中已经到达但由于某些事件而不再需要的信息可以调用 pvm\_freebuf() 破坏掉，使得它不再占据缓冲区空间。在大多数情况下用户并不需使用多个发送和接收缓冲区，用户只须简单地调用 pvm\_initsend() 来重清发送缓冲区即可。

当使用多缓冲区时，通常是对每一个被组装的信息先创建一个缓冲区，然后再释放。事实上 pvm\_initsend() 就是对缺省的发送缓冲区先调用一次 pvm\_freebuf() 再调用一次 pvm\_mkbuf()。

错误信息：

pvm\_freebuf 返回如下错误码：

错误名	可能的原因
PvmBadParam	参数不合法
PvmNoSuchBuf	bifid 值不合法
(4) pvm_getsbuf(), pvmfgetbuf()	

调用格式：

```
int bufid = pvm_getsbuf (void)
call pvmfgetbuf (bufid)
```

功能：返回活跃的发送缓冲区标识。

描述：

pvm\_getsbuf() 返回活跃的发送缓冲区标识，若当前没有活跃的发送缓冲区，则

返回 0。

错误信息：

本调用不返回任何错误信息。

(5) pvm\_getrbuf( ), pvmfgetrbuf( )

调用格式：

```
int bufid = pvm_getrbuf (void)
call pvmfgetrbuf (bufid)
```

功能：返回活跃的接收缓冲区标识。

描述：

pvm\_getrbuf( ) 返回活跃的接收缓冲区标识，若当前没有活跃的接收缓冲区，则返回 0。

错误信息：

本调用不返回任何错误信息。

(6) pvm\_setsbuf( ), pvmfsetsbuf( )

调用格式：

```
int oldbuf = pvm_setsbuf (int bufid)
call pvmfsetsbuf (bufid, oldbuf)
```

功能：切换活跃发送缓冲区。

参数：

bufid：用做新的发送缓冲区的缓冲区标识。

oldbuf：返回值，存放以前的活跃发送缓冲区标识号。

描述：

pvm\_setsbuf( ) 将活跃发送缓冲区切换到由 bufid 指定的缓冲区，并保存以前的活跃发送缓冲区，其缓冲区标识返回在 oldbuf 中。若 bufid 置为 0，则保存当前活跃发送缓冲区，此时不再存在活跃发送缓冲区。

当管理多缓冲区时需要使用该子程序。比如，在两个缓冲区之间切换。一个缓冲区用来发送信息给图形接口，另一个缓冲区用来发送数据给应用软件的其它任务。

错误信息：

pvm\_setsbuf 返回如下错误码：

错误名	可能的原因
PvmBadParam	参数不合法
PvmNoSuchBuf	bufid 值不合法

(7) pvm\_setrbuf( ), pvmfsetrbuf( )

调用格式：

```
int oldbuf = pvm_setrbuf (int bufid)
call pvmfsetrbuf (bufid, oldbuf)
```

功能：切换活跃接收缓冲区。

参数：

bufid：用做新的接收缓冲区的缓冲区标识。

`oldbuf`: 返回值, 存放以前的活跃接收缓冲区标识号。

描述:

`pvm_setrbuf()` 将活跃接收缓冲区切换到由 `bufid` 指定的缓冲区, 并保存以前的活跃接收缓冲区。其缓冲区标识返回在 `oldbuf` 中。若 `bufid` 置为 0, 则保存当前活跃接收缓冲区, 此时不再存在活跃接收缓冲区。

当管理多缓冲区时需要使用该子程序。

错误信息:

`pvm_setrbuf` 返回如下错误码:

错误名	可能的原因
PvmBadParam	参数不合法
PvmNoSuchBuf	<code>bufid</code> 值不合法

## 2. 装填数据

下列这些 C 例程都用于把一个给定数据类型的数组装填到一个活跃的发送缓冲区中, 它们可被多次调用以装填一组信息, 而这组信息可以包含多个不同数据类型的数组。PVM 对信息的复杂性并没有限制, 但一个应用程序必须准确地按照装填方式卸出数据, 要传送一个 C 的结构, 必须依次装填其各个成员数据。

每个装填历程都有以下几个参数: 一个指向即将装填的第一个元素的指针, `nitem` 指明即将装填的数组中元素的个数, `stride` 指明装填时的步长, 例程 `pvm_pkstr()` 式个例外, 它用于装填一个 NULL 结尾的字符串, 而不用 `nitem` 和 `stride` 两个参量。

- `pvm_pk *` (), `pvmfpack` ()

调用格式:

```
int info = pvm_pkbyte (char * xp, int nitem, int stride)
int info = pvm_pkcplx (float * cp, int nitem, int stride)
int info = pvm_pkdcplx (double * zp, int nitem, int stride)
int info = pvm_pkdouble (double * dp, int nitem, int stride)
int info = pvm_pkfloat (float * fp, int nitem, int stride)
int info = pvm_pkint (int * ip, int nitem, int stride)
int info = pvm_pklong (int * ip, int nitem, int stride)
int info = pvm_pkshort (int * jp, int nitem, int stride)
int info = pvm_pkstr (char * sp)
call pvmfpack (what, xp, nitem, stride, info)
```

功能: 将用户提供的各种类型的数据组装到活跃信息缓冲区中。

参数:

`nitem`: 要组装的数据项数目。

`stride`: 组装数据项时的间隔。

`xp`: 指向一组字节的指针, 这一组字节所存放的数据可以是任何数据类型, 但数据提取时也必须是相应的数据类型。

`cp`: 复数数组, 至少有 `nitem * stride` 个元素。

`xp`: 双精度复数数组, 至少有 `nitem * stride` 个元素。

dp: 双精度实数数组, 至少有 nitem \* stride 个元素。

fp: 实数数组, 至少有 nitem \* stride 个元素。

ip: 整数数组, 至少有 nitem \* stride 个元素。

jp: 整数 \* 2 数组, 至少有 nitem \* stride 个元素。

sp: 指向一个以 null 结束的字符串的指针。

what: 一个整数, 用来指定被组装的数据类型, 其可能的取值如下:

STRING	0
BYTE1	1
INTEGER2	2
INTEGER4	3
REAL4	4
COMPLEX	5
REAL8	6
COMPLEX16	7

info: 返回的整数状态码, 负值表示出错。

描述:

每一个 pvm\_pk \* () 子程序将一种由特定数据类型的数据构成的数组组装到活跃发送缓冲区中。对于这些子程序, 第一个参数是指向要组装的第一个数据项的指针, 要组装的数据项数目为 nitem, 组装时从数组中取数据项的间隔为 stride。pvm\_pkstr () 是一个例外, 它是组装一个由 NULL 结束的字符串, 所以它不需要 nitem 和 stride 参数, Fortran 例程 pvmfpack (STRING, ...) 要求 nitem 为字符串中字符个数, stride 为 1。

若组装成功, info 返回 0, 否则返回负值。

单个变量(不是数组)可以通过设置 nitem=1 和 stride=1 来组装。C 语言中的结构变量必须一次一个元素地组装。

当 XDR 支持 64 位的数据类型时, 今后将扩充 what 参数, 使之包括 64 位的数据类型。但到那时, 用户也应意识到从 CRAY 这样的 64 位机器到 SPARC 工作站这样的 32 位机器传送数据时可能的精度损失。即使 PVM 的配置是异构的, 也可以设置数据格式转换方法为 PvmDataRaw, 来保证在两个 64 位机器之间传送数据不损失精度。

错误信息:

pvm\_pk \* 返回如下错误码:

错误名	可能的原因
PvmNoMem	信息缓冲区尺寸超过了该机上可用的内存
PvmNoBuf	没有活跃的发送缓冲区

### 3. 发送和接收数据

#### (1) pvm\_send (), pvmfsend ()

调用格式:

```
int info = pvm_send (int tid, int msgtag)
call pvmfsend (tid, msgtag, info)
```

功能：发送在活跃发送缓冲区中的信息

参数：

`tid`：信息接收者的 `tid`。

`msgtag`：指定要接收信息的类型。

`info`：返回的整数状态码，负值表示出错。

描述：

`pvm_send()` 将存放在活跃发送缓冲区的信息发送给由 `tid` 指定的 PVM 进程。`msgtag` 为该信息的类型。若调用成功，`info` 返回 0，否则返回负值。`pvm_send()` 是异步的，只要信息安全地向接收者发送，该调用即返回，这与同步通讯不同，在同步通讯中，发送者阻塞直到接收者都完成相应的接收为止。

错误信息：

`pvm_send` 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法
PvmNoBuf	没有活跃的发送缓冲区

### (2) `pvm_recv()`, `pvmfrecv()`

调用格式：

```
int bufid = pvm_recv (int tid, int msgtag)
call pvmfrecv (tid, msgtag, bufid)
```

功能：阻塞信息接收，阻塞调用任务直至从指定发送者发送来的指定类型的信息到达为止，并将接收到的信息存放在一个新的活跃接收缓冲区中。

参数：

`tid`：用户指定的要接收信息的发送者 `tid`，-1 为通配符，匹配任何发送者。

`msgtag`：指定要接收信息的类型。`msgtag` 应大于 0；`msgtag`=-1 表示匹配任何信息类型。

`bufid`：返回活跃的接收缓冲区标识号，负值表示出错。

描述：

所谓阻塞是指当接收信息尚未到达时，该子程序不会返回。当要接收的信息到达后，它才返回，一旦返回，就可以调用数据提取子程序来将信息缓冲区中的数据项提取到用户存储区。

错误信息：

`pvm_recv` 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法

### (3) `pvm_mcast()`, `pvmfmcast()`

调用格式：

```
int info = pvm_mcast (int *tids, int ntask, int msgtag)
```

---

```
call pvmfmcast (ntask, tids, msgtag, info)
```

功能：将活跃发送缓冲区中的数据广播给一组任务。

参数：

ntask：接收数据的任务数。

tids：一个具有 ntask 个元素的整型数组，它的每一个元素中存放一个接收数据的 PVM 进程的 tid。

msgtag：指定要发送的信息类型，msgtag 应大于 0。

info：返回值，若为负值则表示出错。

描述：

pvm\_mccast（）将活跃发送缓冲区中的数据广播给 ntask 个由数组 tids 指定的任务。若发送任务的 tid 也在 tids 数组中，则信息也发送给自身。若调用成功，info 返回 0，否则返回负值。

接收进程通过 pvm\_recv（）和 pvm\_nrecv（）来接收广播发送的信息。pvm\_mccast（）是异步的，基于 pvmd 之间的最小分叉树算法。这与同步通讯不同，在同步通讯中，发送者阻塞直到所有的接收者都完成相应的接收为止，pvm\_mccast（）首先决定哪些 pvmd 包含指定的任务，对这些 pvmd 进行基于分叉树的广播，然后这些 pvmd 再将信息传递给它们的本地任务。

错误信息：

pvm\_mccast 返回如下错误码：

错误名	可能的原因
PvmSysErr	Pvmd 不响应
PvmBadParam	参数不合法
PvmNoBuf	没有活跃的发送缓冲区

(4) pvm\_nrecv（），pvmfnrecv（）

调用格式：

```
int bufid = pvm_nrecv (int tid, int msgtag)
call pvmfnrecv (tid, msgtag, bufid)
```

功能：非阻塞的信息接收。

参数：

tid：用户指定的要接收信息的发送者 tid，-1 为通配符，匹配任何发送者。

msgtag：指定要接收信息的类型。msgtag 应大于 0；msgtag=-1 表示匹配任何信息类型。

bufid：返回活跃的接收缓冲区标识号。负值表示出错；0 表示所要接收的信息尚未到达。

描述：

pvm\_nrecv（）检查由 tid 指定的发送者发送的信息类型为 msgtag 的信息是否已经到达。如果到达，立即将它存放到新的活跃发送缓冲区，并返回其缓冲区标识到 bufid，如果有当前接收缓冲区的话，它还清除该缓冲区。如果要接收的信息尚未到达，它也立即返回，此时 bufid 返回 0，若出错则返回负值。

tid 或 msgtag 的 -1 均表示通配符，这给用户提供了很大的灵活性。如果用户指定

信息类型 msgtag 而 tid = -1，则接收从任何任务发送来的具有指定类型的信息。如果用户只指定 tid 而 msgtag = -1，则接收从指定任务来的任何类型的信息。若 tid 和 msgtag 均为 -1，则接收从任何任务来的任何信息。

所谓非阻塞的是指不管要接收的信息到达与否，该子程序总是马上返回。此时或者接收到信息，或者是给出一个状态信息表示指定的信息尚未到达。可以多次调用 pvm\_nrecv() 来查看一个指定的信息是否已经到达。此外，如果应用软件在接收数据以前已经完成了它的全部工作，它可以调用 pvm\_recv() 来接收信息。若接收到信息，则可以调用数据提取子程序来将缓冲区中的数据提取到用户存储区中。

错误信息：

(5) pvm\_nrecv 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法

(6) pvm\_probe(), pvmfprobe()

调用格式：

```
int bufid = pvm_probe (int tid, int msgtag)
call pvmfprobe (tid, msgtag, bufid)
```

功能：检测指定信息是否已经到达。

参数：

tid：用户指定的要接收信息的发送者 tid，-1 为通配符，匹配任何发送者。

msgtag：指定要接收的信息类型。msgtag 应大于 0；msgtag = -1 表示匹配任何信息类型。

bufid：返回活跃的接收缓冲区标识号，负值表示出错，0 表示所要接收的信息尚未到达。

描述：

pvm\_probe 检查由 tid 指定的发送者发送的信息类型为 msgtag 的信息是否已经到达。如果到达，返回其接收缓冲区标识到 bufid，用户再调用 pvm\_bufinfo() 时可以使用 bufid 来获得该信息的有关信息。如果要接收的信息尚未到达，它也立即返回，此时 bufid 返回 0，若出错则返回负值。

tid 或 msgtag 的 -1 均表示通配符，这给用户提供了很大的灵活性。如果用户指定信息类型 msgtag 而 tid = -1，则接收从任何任务发送来的具有指定类型的信息。如果用户只指定 tid 而 msgtag = -1，则接收从指定任务来的任何类型的信息。若 tid 和 msgtag 均为 -1，则接收从任何任务来的任何信息。

可以多次调用 pvm\_probe() 来查看一个指定的信息是否已经到达。在信息到达后，必须先调用 pvm\_recv()，然后才可以调用数据提取子程序来将缓冲区中的数据提取到用户存储区中。

错误信息：

pvm\_probe 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应

PvmBadParam      参数不合法

(7) pvm\_bufinfo( ), pvmfbufinfo( )

调用格式:

```
int info = pvm_bufinfo (int bufid, int *bytes, int *msgtag, int *tid)
```

```
call pvmfbufinfo (bufid, bytes, msgtag, tid, info)
```

功能: 返回指定信息缓冲区的有关信息。

参数:

bufid: 信息缓冲区的标识。

bytes: 一个整型量, 返回信息缓冲区的大小, 以字节为单位。

msgtag: 一个整型量, 返回指定信息缓冲区中信息的类型, 若用 pvm\_recv( ) 接收信息时信息类型是通配符, 则信息接收者可利用此来得到所接收信息的确切类型。

tid: 一个整型量, 返回指定信息缓冲区中信息的发送者, 若用 pvm\_recv( ) 接收信息时信息发送者是通配符, 则信息接收者可利用此来得到所接收信息的确切发送者。

描述:

pvm\_bufinfo 返回指定信息缓冲区的有关信息。通常用它来确定最后接收到信息的有关信息, 如信息大小、发送者、信息类型等。有些应用软件设计成可以接收任何信息, 然后根据接收到的信息发送者和类型来采取不同的动作, 此时, 该子程序特别有用。若 pvm\_bufinfo 调用成功, info 返回 0, 否则返回负值。

错误信息:

pvm\_bufinfo 返回如下错误码:

错误名	可能的原因
PvmNoBuf	没有活跃的接收缓冲区

(8) pvm\_advise( ), pvmfadvise( )

调用格式:

```
int info = pvm_advise (int route)
```

```
call pvmfadvise (route, info)
```

功能: 建议 PVM 是否使用直接的任务到任务的路由。

参数:

route: 一个整数, 建议 PVM 是否建立直接的任务到任务的连接。

route 的取值如下:

PvmDontRoute 1 不允许与该任务建立直接的连接

PvmAllowDirect 2 允许但不要求与该任务建立直接的连接

PvmRouteDirect 3 要求与该任务建立直接的连接

描述:

pvm\_advise( ) 建议 PVM 是否对所有的后继通信建立直接的任务到任务的连接(使用 TCP)。连接一旦建立, 就一直存在, 直至应用程序完成。如果要建立连接的两个任务中的一个要求 PvmDontRoute, 或系统资源用完了, 则不能建立直接的连接, 此

时采用 Pvmdaemon 的缺省路由。pvm\_advise( ) 可以调用多次来选择性地建立直接连接，但通常都是在每个任务开头时调用一次。PvmAllowDirect 是缺省设置。任务 A 设置为 PvmAllowDirect 就允许其它任务建立到任务 A 的直接连接，一旦在两个任务之间建立好了直接连接，这两个任务将利用它来传送信息。直接的任务到任务的连接，其性能可以比缺省的路由高两倍，其缺点是缺乏可扩展性。有些 UNIX 版本限制连接数不超过 30。

错误信息：

pvm\_probe 返回如下错误码：

错误名	可能的原因
PvmBadParam	参数 route 不合法

#### 4. 取出数据

下列这些 C 例程都用于从接收缓冲区中取出数据。一个应用程序必须准确地按照装填方式卸出数据，在数据类型、元素个数、步长等方面都准确匹配，一个指向即将取出的第一个元素的指针，nitem 指明即将取出的数组中元素的个数，stride 指明取出时的步长，例程 pvm\_upkstr( ) 式例外，它用于装填一个 NULL 符串，而不用 nitem 和 stride 两个参量。

- pvm\_upk \* ( ), pvmunpack( )

调用格式：

```
int info = pvm_upkbyte (char * xp, int nitem, int stride)
int info = pvm_upkcomplex (float * cp, int nitem, int stride)
int info = pvm_upkdouble (double * dp, int nitem, int stride)
int info = pvm_upkfloat (float * fp, int nitem, int stride)
int info = pvm_upkint (int * ip, int nitem, int stride)
int info = pvm_upklong (int * ip, int nitem, int stride)
int info = pvm_upkshort (int * jp, int nitem, int stride)
int info = pvm_upkstr (char * sp)
call pvmunpack (what, xp, nitem, stride, info)
```

功能：从信息接收缓冲区中提取指定数据类型的数据项。

参数：

nitem：要提取的数据项数目。

stride：提取数据项时的间隔。

xp：指向一组字节的指针，这一组字节所存放的数据可以是任何数据类型，但数据提取时也必须是相应的数据类型。

cp：复数数组，至少有 nitem \* stride 个元素。

xp：双精度复数数组，至少有 nitem \* stride 个元素。

dp：双精度实数数组，至少有 nitem \* stride 个元素。

fp：实数数组，至少有 nitem \* stride 个元素。

ip：整数数组，至少有 nitem \* stride 个元素。

jp: 整数 \* 2 数组, 至少有 nitem \* stride 个元素。

sp: 指向一个以 NULL 结束的字符串。

what: 一个整数, 用来指定被提取的数据类型, 其可能的取值如下:

STRING	0
BYTE1	1
INTEGER2	2
INTEGER4	3
REAL4	4
COMPLEX	5
REAL8	6
COMPLEX16	7

info: 返回的整数状态码, 负值表示出错。

描述:

每一个 pvm\_upk \* () 子程序将活跃接收缓冲区中的数据按指定数据类型提取到用户存储区中。对于这些子程序, 第一个参数是指向要提取的第一个数据项的指针, 要提取的数据项数目为 nitem, 提取时从数组中取数据项的间隔为 stride。pvm\_upkstr() 是一个例外, 它是提取一个由 NULL 结束的字符串, 所以它不需要 nitem 和 stride 参数, FORTRAN 例程 pvmfunpack(STRING, ...) 要求 nitem 为字符串中字符个数, stride 为 1。

若提取成功, info 返回 0, 否则返回负值。

单个变量(不是数组)可以通过设置 nitem=1 和 stride=1 来提取。C 语言中的结构变量必须一次一个元素的提取。

从数据缓冲区中提取数据应该严格地跟组装时匹配, 以保证数据的一致性。

错误信息:

pvm\_upk \* 返回如下错误码:

错误名	可能的原因
PvmNoData	读到了接收缓冲区尾; 最大可能是试图提取的数据项比组装的数据项多
PvmNoBuf	没有活跃的接收缓冲区
PvmBadMsg	无法对所接收的信息完成数据格式转换

#### § 11.4.7 动态进程组类函数

动态进程组的功能建立在核心 PVM 函数上, 用户程序中要使用这些功能必须与一个独立的库 libgpvm3.a 连接, pvm3 并不直接处理进程组功能。当用户程序中出现了有关进程组的调用时, 将会自动启动一个进程组服务器, 来完成有关功能。

明确的进程组功能是 PVM3 的一大特色, 为了跟 PVM 的初衷一致, 进程组功能被设计的非常通用并且在一些宿主机上对用户透明, 任何 PVM 任务在任何时候都可加入或离开一个组, 而不会影响在相关组里的别的任务。一个任务可以向一个不包含它的组中所有任务广播数据。更简单地说: 除了 pvm\_lvgroup() 和 pvm\_barrier() 要求调用者必须

已经是某个组的成员，任何 PVM 任务在任何时候都可以进行下面的有关进程组的调用。

(1) pvm\_joingroup( ), pvmfjoingroup( )

调用格式：

```
int inum = pvm_joingroup (char * group)
call pvmfjoingroup (group, inum)
```

功能：将调用进程增加到指定的进程组中。

参数：

group：存放进程组组名的字符串。  
inum：返回调用进程在进程组中的实例号。

描述：

pvm\_joingroup( ) 将调用进程加入到由 group 指定的进程组中，它返回进程在该进程组中的实例号。若出错 inum 返回负值。实例号从 0 开始依次递增。当使用进程组时，(group, inum) 唯一标识一个 PVM 进程，这与以前的 PVM 命名方法一致。

如果一个任务调用 pvm\_lvgroup( ) 脱离一个进程组，然后再调用 pvm\_joingroup( ) 重新加入该进程组，不能保证它会得到相同的实例号。当一个任务加入一个进程组时，它总是得到最小可用的实例号。在 PVM 中，一个任务可以同时属于多个进程组。

错误信息：

pvm\_joingroup 返回如下错误码：

错误名	可能的原因
PvmSysErr	Pvmd 不响应
PvmBadParam	参数不合法
PvmDupGroup	企图重复进入一个组

(2) pvm\_lvgroup( ), pvmflvgroup( )

调用格式：

```
int info = pvm_lvgroup (char * group)
call pvmflvgroup (group, info)
```

功能：调用进程脱离指定的进程组。

参数：

group：存放进程组组名的字符串。  
info：返回状态码。

描述：

pvm\_lvgroup( ) 使调用进程脱离指定的进程组，若调用成功，info 返回 0。若出错 info 返回负值。

错误信息：

pvm\_lvgroup 返回如下错误码：

错误名	可能的原因
PvmSysErr	Pvmd 不响应
PvmBadParam	参数不合法
PvmNoGroup	由 group 指定的进程组不存在

PvmNotInGroup 调用进程不是该进程组成员

(3) pvm\_gettid(), pvmfgettid()

调用格式：

```
int tid = pvm_gettid (char *group, int inum)
call pvmfgettid (group, inum, tid)
```

功能：给出进程组名和实例号，获取对应的 PVM 进程的 tid。

参数：

group：存放进程组组名的字符串。

inum：进程在进程组中的实例号。

tid：返回进程的 tid。

描述：

pvm\_gettid() 返回由 group 指定的进程组中实例号为 inum 的 PVM 进程的 tid。

若调用成功，tid 大于 0。否则返回负值。

错误信息：

pvm\_gettid 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法
PvmNoGroup	由 group 指定的进程组不存在
PvmNoInst	进程组中没有该实例

(4) pvm\_getinst(), pvmfgetinst()

调用格式：

```
int inum = pvm_getinst (char *group, int tid)
call pvmfgetinst (group, tid, inum)
```

功能：给出进程组名和 tid，获取对应的实例号。

参数：

group：存放进程组组名的字符串。

tid：PVM 进程的 tid。

inum：返回进程在进程组中的实例号。

描述：

pvm\_getinst() 返回由 group 指定的进程组中 PVM 进程号为 tid 的实例号到 inum，若调用成功，inum 大于或等于 0，否则返回负值。

错误信息：

pvm\_getinst 返回如下错误码：

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法
PvmNoGroup	由 group 指定的进程组不存在
PvmNotInGroup	进程不是该进程组成员

## (5) pvm\_gsize(), pvmfgsize()

调用格式:

```
int size = pvm_gsize (char * group)
call pvmfgsize (group, size)
```

功能: 返回指定进程组中的成员数。

参数:

group: 存放进程组组名的字符串。

size: 返回进程组中的成员数, 负值表示出错。

描述:

pvm\_gsize() 返回由 group 指定的进程组中进程数。若调用失败 size 为负值。

因为在 PVM3.0 中, 进程组可以动态改变, 所以该子程序只保证返回指定进程组某个时刻的成员数。

错误信息:

pvm\_gsize 返回如下错误码:

错误名	可能的原因
PvmSysErr	pvmd 不响应
PvmBadParam	给出的组名不合法

## (6) pvm\_barrier(), pvmfbarrier()

调用格式:

```
int info = pvm_barrier (char * group, int count)
call pvmfbarrier (group, count, info)
```

功能: 阻塞调用进程直至同组中的 count 个进程均调用此子程序为止。

参数:

group: 存放进程组组名的字符串。

count: 一个整数, 指定参与同步的进程数。

info: 返回状态码。

描述:

pvm\_barrier() 阻塞调用进程, 直至由 group 指定的进程组中的 count 个进程调用 pvm\_barrier 为止。要求提供 count 参数是因为在其它进程调用 pvm\_barrier() 以后, 可能还有进程动态地加入指定的进程组, 这样, 在任一给定时候, pvm 并不知道有多少成员进程在等待同步。当然 count 可以设置的较小, 但通常是设置为该组中成员进程的总数。所以, pvm\_barrier() 提供了一个进程组内进程同步的手段。在任何一次 barrier 同步期间, 所有参与同步的成员进程必须以相同的 count 值调用 pvm\_barrier()。一旦一个指定的 barrier 会合点成功地通过, 该进程组的成员进程又可以用同一组名再次调用 pvm\_barrier()。

如果 count 值指定为 -1, 则 PVM 使用 pvm\_gsize() 值作为 count 值。在一个进程组建立以后不再变化的应用程序中, 这种用法十分有用。

若 pvm\_barrier() 调用成功, info 返回 0, 否则返回负值。

错误信息:

pvm\_barrier 返回如下错误码:

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法
PvmNoGroup	由 group 指定的进程组不存在
PvmNotInGroup	进程不是该进程组成员

#### (7) pvm\_bcast ( ), pvmfbcast ( )

调用格式:

```
int info = pvm_bcast (char * group, int msgtag)
call pvmfbcast (group, msgtag, info)
```

功能: 立即广播活跃发送缓冲区中的数据。

参数:

group: 存放进程组组名的字符串。

msgtag: 一个整数, 指定发送的数据类型。

info: 返回状态码。

描述:

pvm\_bcast ( ) 将存放在活跃发送缓冲区中的信息广播给由 group 指定的进程组中的所有进程, 若调用任务本身也是该进程组中的成员, 在 PVM3.0 和 PVM3.1 中, 信息也会发送给自身。在 PVM 的下一个版本中, 被广播的信息不会再发送给广播者。任何 PVM 任务都可以调用 pvm\_bcast ( ), 它不必是 group 指定的进程组的成员。信息的内容通过 msgtag 来区分。若调用成功, info 返回 0, 否则返回负值。

pvm\_bcast ( ) 是异步的, 只要信息安全地开始向接收者传送, 调用即返回, 这与同步通讯不同, 在同步通信中, 发送者阻塞直至所有的接收者都完成相应的接收为止。

pvm\_bcast ( ) 首先检查进程组数据库来找出成员进程的 tid, 然后根据这些 tid 对全组进程完成广播。如果广播期间进程组发生了变化, 还在进行的广播不受该变化的影响。

错误信息:

pvm\_bcast 返回如下错误码:

错误名	可能的原因
PvmSysErr	pvmrd 不响应
PvmBadParam	参数不合法
PvmNoGroup	由 group 指定的进程组不存在

## § 11.5 PVM 应用实例

征得孙家昶教授同意, 我们从他们的著作《网络并行计算与分布式编程环境》中摘录了下面的 PVM 应用实例。(限于篇幅, 略去程序。)

### 1. 问题及算法

考虑下述定义在三维长方体  $\Omega = (0, A) \times (0, B) \times (0, C)$  上满足 Dirichlet 边界

条件的 Poisson 方程：

$$\begin{cases} -\Delta u = f \\ u|_{\partial\Omega} = u_0 \end{cases}$$

假设  $x, y, z$  分别代表三个空间方向，我们在  $N_x \times N_y \times N_z$  的均匀网格上采用最简单的 7 点差分格式对上述方程进行离散，三个方向的网格步长分别为

$$h_x = A/N_x, \quad h_y = B/N_y, \quad h_z = C/N_z$$

离散后得到下述差分方程组：

$$\begin{cases} a_0 u_{i,j,k} + a_x (u_{i+1,j,k} + u_{i-1,j,k}) + a_y (u_{i,j+1,k} + u_{i,j-1,k}) + a_z (u_{i,j,k+1} + u_{i,j,k-1}) = f_{i,j,k} \\ i=1, \dots, N_x-1, \quad j=1, \dots, N_y-1, \quad k=1, \dots, N_z-1 \end{cases}$$

其中  $u_{i,j,k} = u(ih_x, jh_y, kh_z)$ ,  $f_{i,j,k} = f(ih_x, jh_y, kh_z)$ . 差分方程中的系数由下列公式给出：

$$a_0 = \frac{1}{h_x^2} + \frac{2}{h_y^2} + \frac{2}{h_z^2}, \quad a_x = -\frac{1}{h_x^2}, \quad a_y = -\frac{1}{h_y^2}, \quad a_z = -\frac{1}{h_z^2}$$

由于是线性问题，我们采用多重网格法中的 CS (Correction Storage) 算法求解上面的问题，使用的网格重数  $l$  取使得  $N_x, N_y, N_z$  能同时被  $2^{l-1}$  整除的最大整数。粗网格步长通过将细网格步长依次乘 2 得到。显然，当  $N_x, N_y, N_z$  中有一个为奇数时算法退化为单网格方法。基本迭代算法采用红黑顺序 Gauss-Seidel 迭代法，它的收敛速度远远高于自然顺序 Gauss-Seidel 迭代法或 Jacobi 迭代法，并且对于所用的七点差分格式其迭代过程可完全并行化。

为在具有  $P = P_x \times P_y \times P_z$  个处理器的分布式系统上实现上述算法，我们将计算区域划分成  $P_x \times P_y \times P_z$  个长方体形子区域，每个子区域中的变量存储在一个处理器中，相应的计算任务也由该处理器来完成。计算过程中每两个相邻的子区域间需要进行通信交换数据。显然，当总处理器个数  $P$  固定时，应该选取  $P_x, P_y, P_z$  使得  $(P_x-1)/(N_x-1) + (P_y-1)/(N_y-1) + (P_z-1)/(N_z-1)$  尽量小，从而减少总的通信量。

为方便通信与计算，每个处理器中除存储多重网格法所使用的每重网格包含在相应子区域内的网格点上的变量之外，还存有相邻子区域中的一排网格点上的变量。这样，存储在每个处理器中的网格点构成一个子网格，相应的子区域严格包含于该子网格中，并且相邻两个子网格正好有一个网格步长的重叠部分。

为简单起见，我们仅考虑相邻子区域的顶点重叠的情形，即区域的剖分可在三个空间方向上分别进行。以  $x$  方向为例，只需选取  $P_x+1$  个数  $A_0, A_1, \dots, A_{P_x}$ ，它们满足：

$$0 = A_0 < A_1 < \dots < A_{P_x} = A$$

它们便将区间  $[0, A]$  分为  $P_x$  个子区间。假设  $x$  方向上的网格点为  $x_i = ih_x$ ,  $0 \leq i \leq N_x$ ，对任一子区间  $[A_k, A_{k+1}]$  ( $0 \leq k \leq P_x-1$ )，假设存储在对应于该子区间的处理器中的网格点为  $\{x_i | I_k \leq i \leq J_k\}$ ，则可取

$$I_k+1 = \text{满足 } 1 \leq j \leq N_x \text{ 及 } x_j > a_k \text{ 的最小整数}$$

$$J_k-1 = \text{满足 } 0 \leq j \leq N_x-1 \text{ 及 } x_j \leq a_{k+1} \text{ 的最大整数}$$

其他两个空间方向上的处理完全一样。不难看出如此形成的区域剖分具有我们所要求的性质。

在较粗的网格上，某些子网格可能没有内点。这些子网格实际上仅在通信时起传递数

据的作用。

下面分析上述网格部分的一些性质，以便读者了解程序实现的一些细节。

1) 相邻子网格之间正好有一个网格步长的重叠，不同子网格的内点互不相交。因此 Gauss—Seidel 迭代只需在每个子网格的内点上独立进行。根据 7 点差分格式上红黑顺序 Gauss—Seidel 迭代法的性质，所有子区域中的迭代过程可同时进行，因而算法具有很好的并行度。

2) 细网格到粗网格的余量插值可分别在各处理器中同时进行，每个处理器中进行插值所需要的数据均可在该处理器中获得，无需与相邻处理器进行通信。

3) 同样，在将粗网格上的解插值到细网格上对细网格上的解进行修正的过程也可完全分别在各处理器中同时进行，不需要通信。

多重网格算法中的几个主要步骤的具体实现如下：

1) Gauss—Seidel 迭代。每次 Gauss—Seidel 迭代分两步进行。第一步，在所有处理器中的“红”点上进行一次迭代，然后在相邻处理器之间的重叠点上交换新的近似解的值。第二步，在“黑”点上进行一次迭代，然后在相邻处理器之间交换新的近似解的值。

2) 细网格到粗网格的余量插值。可在各处理器上同时进行，无需进行通信。注意到所有黑点上的余量值为 0，可在插值时减少一些计算量。

3) 粗网格到细网格的修正。可在各处理器上同时进行，无需进行通信。插值完成后需在细网格上进行一次通信以交换修正后得到的新的近似解的值，这里只需交换“黑”点上的值，因为“红”点上的值马上会在 Gauss—Seidel 迭代中被改变。

所有计算均可在各处理器上并行进行。处理器之间的通信只在 Gauss—Seidel 迭代过程中以及粗网格修正之后才需要。

这里所介绍的方法只是将串行算法中的计算工作分配到多台处理器上完成，因而计算结果不受处理器个数的影响。不论使用多少处理器，所得到的结果应该与串行程序的结果一样。

## 2. PVM 程序简介

上面所描述的算法的 PVM 程序名为 pvmp3d，它的 Fortran 源程序包括以下几个文件，这些文件的详细内容在附录 B 中给出。

Makefile.	aimk	用于编译、链接
data		数据文件
pvmp3d.cmm		包含公用参数的定义
pvmp3d.f		主要 Fortran 程序
spawn11.f		替代 PVMFSPAWN 子程序
getime.c		计时函数

pvmp3d.cmm 文件中主要包含一些公共参数的定义，及一些公共数据块。该文件被用 INCLUDE 命令插入到 pvmp3d.f 的每个子程序中。文件开头的两段语句用于设定浮点数字长，通过将其中一段语句变成说明语句而留下另一段语句来选择计算时采用单精度还是双精度。程序中定义了一个大数组，计算中所需要的主要数组变量（包括各重网格上的近似解和右端项）均从该数组中分配，该数组的长度由参数 LENUF 定义，它的值取决于网格大小、网格重数及子区域的大小，通常可根据计算机可用内存的大小来给它一个固定的

值，而不必每次计算时修改它。参数 IVCYCLE 用于选择多重网格算法中的循环类型，当 IVCYCLE 取非 0 值时使用 V 型循环，当 IVCYCLE 等于 0 时采用自适应循环（根据余量衰减的情况来决定循环方式）。参数 NP 和 NQ 分别定义多重网格法中的预迭代（pre-relaxation）和后迭代（post-relaxation）的次数。参数 MAXPROC 定义最大允许的任务数。变量 NPROC 给出实际的任务数，也就是所使用的子区域个数，它可被分解成三个空间方向上的子区域数之积，三个空间方向上的子区域数分别存储在变量 NPROCX，NPROCY，NPROCZ 中。变量 ME 存放当前任务的编号（从 0 到 NPROC-1），它也被分解成三个空间方向上的子编号分别存在变量 MEX，MEXY，MEZ 中，有关 PVM 的其它一些变量（MYTID，PARENTID 等等）的含义是明显的。参数 MAXLVL 定义了最大允许的网格重数，而数组 ISTART，IEND，JSTART，JEND，KSTART，KEND 则分别存放各重网格在该任务中的子网格在三个空间方向上的起始和终止网格点号。pvmp3d.cmm 的最后定义了三个变量 CNTADD，CNTMUL，CNTTMP，它们用来统计总计算量（加减法和乘除法的次数）。

spawnll.f 中定义了一个名为 SPAWN11 的函数，该函数被第一个任务调用生成其它 NPROC-1 个任务，它使得所生成的任务尽量均匀地分配在虚拟机的各台机器上（当任务数等于虚拟机中的机器数时，直接调用 PVMFSPAWN 往往会产生调用的机器上有两个任务，而另外一台机器上却没有任务的问题，造成负载的不均衡）。

gettime.c 中定义了一个时间函数 gettime()，可供 Fortran 程序调用。它通过一个三个元素的双精度数组分别返回用户 CPU 时间、系统 CPU 时间和实际流逝的时间（俗称 wall clock time，即墙上时间）。在有些系统如 IBM AIX 上，函数名中的下划线应该去掉。

Makefile.aimk 用于程序的编译与链接。使用时只需键入 aimk 命令便可编译并链接产生可执行文件 pvmp3d，且被自动放在目录 \$HOME/pvm3/bin/\$PVM\_ARCH 中。

data 是运行时的数据文件。pvmp3d 程序运行时自动从 data 中读入有关区域及网格的参数。该文件中第一行给出最细网格的大小，第二行给出长方体计算区域的三个边长。第三行给出近似解余量的收敛准则。第四行给出各空间方向上的子区域数目。第五行定义 PVM 中使用的路由参数（CALL PVMFSETOPT(PVMROUTE,...)），可通过修改它改变 PVM 的信息传递模式，优化信息传递效率。最后一行给出数据打包时的编码方式，即 encoding 参数，在同构机构成的虚拟并行机上可用 1 (PVMRAW)，而在异种机构成的虚拟并行机上必须用 0 (XDR)。

pvmp3d 程序的主要部分包含在文件 pvmp3d.f 中。该程序的结构属 SPMD 型，运行时一共有 NPROC 个任务，其中 NPROC-1 个任务由第一个任务调用函数 SPAWN11 创建。下面重点解释一下该程序中的主要部分。

程序一开始调用 PVMFMYTID 和 PVMFPARENT 获得自己的 TID 及父进程的 TID。如果父进程的 TID < 0，则表明该任务是由用户启动的第一个任务，于是程序从数据文件 data 中读入所需的参数，计算总任务数 NPROC，调用函数 SPAWN11 启动其它 NPROC-1 个任务，并将运行参数广播给其它任务。如果父进程的 TID ≥ 0，则表明该任务由第一个任务（它的父进程）生成，因此程序接收从第一个任务发送来的运行参数。

当上述步骤完成后，程序在 TIDS 数组中找到对应于自己的 TID 的项，并将该项在数组中的序号做为自己的任务编号（从 0 到 NPROC-1），然后调用 PVMFSETOPT 来设置

PVMROUTE 参数的值。

初始化阶段的最后一个步骤是调用函数 INITMG 计算出多重网格算法所使用的网格重数(NLVLS)、每重网格上分配给自己的子网格在各空间方向的起始和终止网格点号(ISTART, IEND, JSTART, 等等)及所需数组的总长度，并判断由 LENUF 所定义的数组长度是否够用。这些工作完成后，由第一个任务打印出信息“Setup complete”，表明完成了初始化过程。

子程序 SUB 首先初始化近似解 U 及右端项 F，为便于比较，该程序中使用一个解析解做为算例，所需的边界条件和右端项通过解析解得出，而初始近似解则取为 0，然后调用多重网格程序 POIS3D 求出近似解，并与解析解进行比较计算出近似解的误差。在计算过程中，程序自动统计出总的计算量以及所花费的 CPU 时间。

多重网格求解的主要过程在子程序 POIS3D 中完成。由于源程序中穿插有较详细的说明语句，我们不在此对程序做过多的说明。表 11.4 列出了 POIS3D 所用到的子程序、函数名以及它们的作用。

表 11.4 三维 Poisson 方程多重网格算法中的主要子程序及函数

函数或子程序名	用途
FUNCTION CHKCOND	用于各任务间逻辑判断的同步 <sup>1)</sup>
SUBROUTINE MULTIG	红黑顺序 Gauss-Seidel 迭代
FUNCTION EVALRES	计算并返回近似解的余量 <sup>2)</sup>
SUBROUTINE MULTRE	细网格到粗网格的插值（限制算子）
SUBROUTINE MULTPR	粗网格到细网格的插值（扩展算子）
SUBROUTINE XCHGDATA	与相邻子区域交换新近似解的值
SUBROUTINE XCHGDO	被 XCHGDATA 调用

```

TIDS (2 task (s), NPROCX=1 NPROCY=1 NPROCZ=2)
    TID= 4000D SPEED= 1000
    TID= 8000C SPEED= 1000
N= 128 M=128 L=128
A= 1. 0000    B= 1. 0000    C=1. 0000    TOL= 1. 00000E-04
Single Precision, pvm_set; opt (PVMROUTE, 3, IOLD), Encoding=1
U+F size required by this task= 9884K bytes

128 128 128 ( 0, 128) ( 0, 128) ( 0, 65)
 64   64   64 ( 0, 64) ( 0, 64) ( 0, 33)
 32   32   32 ( 0, 32) ( 0, 32) ( 0, 17)
 16   16   16 ( 0, 16) ( 0, 16) ( 0, 9)
   8     8   ( 0,  8) ( 0,  8) ( 0,  5)
   4     4   ( 0,  4) ( 0,  4) ( 0,  3)
   2     2   ( 0,  2) ( 0,  2) ( 0,  2)

=====
Setup complete
=====

128 128 128 ..... it #: 1 Res= 5. 0691328
128 128 128 ..... it #: 2 Res= 0. 41770193
128 128 128 ..... it #: 3 Res= 3. 79756503E-02
128 128 128 ..... it #: 4 Res= 3. 25123221E-03
128 128 128 ..... it #: 5 Res= 2. 97911698E-04
128 128 128 ..... it #: 6 Res= 3. 72482136E-05

Error = 7. 2383881E-04
Number of million operations:
Add/Sub: 337. 70      Mul/Div: 214. 13      Total: 551. 83
Processing time for ME= 0 (seconds):
User: 24. 969      System: 0. 68368      Real: 27. 898
Performance (MFLOPS):
/User: 22. 101      /Real: 19. 780

```

图 11-1 三维多重网格程序的输出实例

## 第十二章 Linda

网络 Linda 是 70 年代由美国 Yale 大学 David Gwlernter 等人构想出来，后由美国 Scientific Computing Associates 公司于 1987 年开发的一个并行环境，经过不断改进，已经推出了许多版本。1993 年的网络 Linda 新版本可以在 Network of Suns, IBM RS 6000s, HP/Apollo5, SGI IRISes, DEC station 等系统机上运行，也可进行并行计算。

网络 Linda 可使用户把网络分布式计算机系统上的所有存储器看作是一个单一的共享存储器或共享 tuple 空间，系统中的任何处理器都可访问其数据，任务和结果。网络 Linda 可分为 C-Linda 和 Fortran-Linda，两个系统的原理和功能相同，但基础语言不一样。

### § 12.1 C-Linda

C-Linda 是一个基于 C 语言的并行程序设计语言，它能够使得用户在宽松的计算机环境下设计并行程序。C-Linda 既可以用于并行已存在的串行应用程序，又可用于开发新的并行应用程序。C-Linda 由并列语句 Linda 和程序设计语言 C 组成。用 C-Linda 设计程序是调用大量的独立的计算过程，组成单个并行程序。这些单个的过程是用 C 语言编写的，用 Linda 把它们胶合在一起。

C-Linda 具有区别于其它语言的并行程序设计环境的特性：

- ① C-Linda 扩充了 C 语言，它不是替换 C 语言，C-Linda 是建立在 C 语言基础之上的；
- ② C-Linda 语言是可移植的，它适合于大量的并行计算机系统，包括共享内存计算机，分布式存储器计算机和网络等，为一个计算机系统编写的 C-Linda 程序，不用修改便可在另外的计算机系统上运行；
- ③ 用 C-Linda 设计程序是简便的；概念上，C-Linda 通过一个逻辑上的全局存储器（虚拟共享存储器）称作“Tuple 空间”和几个简单的操作命令实现并行。

C-Linda 编译程序支持所有的常用开发程序特性，包括编译时间错误检查、实时调试和可视化特性，使用 C-Linda 要求用户有 C 语言知识和并行算法方面的知识。

#### § 12.1.1 Tuple 空间的数据结构

C-Linda 提供一个虚拟共享存储空间—Tuple 空间。C-Linda 用户不用关心 Tuple 空间如何设置，它的物理位置，以及进程之间数据如何通信，所有这些管理是通过 Linda 软件系统完成。C-Linda 在逻辑上有独立于计算机系统的体系结构。C-Linda 程序在不同体系结构的计算机之间是可移植的。它们可以是共享存储器计算机系统、分布式系统或者工作站网络系统等。

从 Tuple 空间存取数据以 Tuple 为单位。Tuple 是 Tuple 结构空间的数据。一个 Tuple

是一个最多为 16 个域的元组，域之间用逗号“,”分开，整个元组用圆括号括起来。Tuple 域主要有下列类型：

整形：int, long, short, char 及 unsigned

实型：float 和 double

结构：struct

联合：union

数组：以上类型的数组，包括多维数组及指针。

例如：

```
int i, * p, a [20], f ();
p=&i;
```

有下列各 Tuple：

(“ simple”, 3) 该 Tuple 有两个域：字符串域和整型常量域；

(“ easy”, i) 该 Tuple 有两个域：字符串域和整型变量 i 域；

(“ integer”, \* p) 该 Tuple 有两个域：字符串域和指针域；

(“ integer”, \* ++a) 该 Tuple 有两个域：字符串域、数组指针域。

### § 12.1.2 C-Linda 对 Tuple 空间的存取操作

C-Lind 提供了四个对 Tuple 空间的基本操作命令和两个操作函数。

- 四个基本操作命令：

out：送一个 Tuple 到 Tuple 空间；

eval：创建一个活动 Tuple（通常创建一个进程）；

in：从 Tuple 空间移出一个 Tuple；

rd：从 Tuple 空间读取一个 Tuple，该 Tuple 仍在 Tuple 空间。

例如：

```
out (“ cube”, 4, 64); 把 Tuple (“ cube”, 4, 64) 送入 C-Linda 空间；
```

eval (“ test”, i, f (i)); 创建一个活动 Tuple (“ test”, i, f (i)), 每一个域为一字符串，第二个域为变量 i 的值，第三个域待定，同时起动进程 f (i); f (i) 最后的结果填入该活动 Tuple 第三个域中。

eval 与 out 的执行过程不同；如下列两个操作命令：

```
eval (“ coord”, x, f (x));
```

```
out (“ coord”, x, f (x));
```

eval 操作在 Tuple 空间建立一个活动 Tuple，起动 f (x)，立即返回执行 eval 后面的语句，不等待；out 操作首先计算 f (x) 的值，f (x) 结束之后在 Tuple 空间写入一 Tuple。该命令没结束之前不执行后续语句。但最终在 Tuple 空间写入的 Tuple 数据相同。

in (“ cube”, 4, ? i); 该操作在 Tuple 空间中配备第一个域为“ Cube”，第二个域为 4，第三个域是整型数的 Tuple。如果配备成功，则在 Tuple 空间消去该 Tuple，并把第三域的值返回给变量 i，如果配备不成功，则等待。

rd 操作类似 in 操作，只是配备成功时，读取该 Tuple，并不在 Tuple 空间消除该 Tuple，例如有语句：

```
for (i=0, i<3, i++) out (" num", i);
```

则在 Tuple 空间有: (" num", 0)、(" num", 1)、(" num", 2) 三个 Tuple, 若有操作:

```
rd (" num", ? i);
```

则 i 返回的值可能是 0, 1, 2 的任何一个, 且该三个 Tuple 仍在 Tuple 空间中。若 rd 改为 in 也可能在该 Tuple 中随机地移出一个 Tuple, 并把该 Tuple 的第二个域的值传给变量 i。

利用 out、eval、in、rd 操作可方便地实现同步并行计算。

- 两个对 Tuple 空间的操作函数: inp 函数和 rdp 函数。

inp、rdp 函数功能类似于 in、rd 命令。当 inp、rdp 配备成功时, 返回值 1, 效果分别与 in、rd 等同; 配备不成功时, 不等待, 返回值 0, 操作无效。利用 inp、rdp 操作替换 in、rd 操作很容易实现异步并行计算(参看本章的例题)。

### 12.1.3 Tuple 配备规则

指定 Tuple 和 Basic Tuple 的配备规则:

- ① 它们包含相同数量的域;
- ② 所有对应有域的相同的类型;
- ③ 所有对应的域有相同的大小(指元素个数);
- ④ 对应的常量域有相同的值。

#### (1) 标量

设有下列语句:

```
int i, * p, a [20], f ();
p=&i;
```

下面的操作全都会在 Tuple 空间存入一个第一个域为字符串" integer", 第二个域为整型值的 Tuple。下例中的注释指第二个域。

```
out (" integer", 3); /* 整型常量 */
out (" integer", i); /* 整型变量 */
out (" integer", * p); /* 指针所指单元的值 */
out (" integer", a [f]); /* 整型数组的数组元素 */
out (" integer", * + + a); /* 指针所指单元的值 */
```

其它类型的标量操作类似。

#### (2) 固定数组

对固定数组, Tuple 的域可用数组名。这里举例说明往 Tuple 空间存一数组和从 Tuple 空间读一数组, 设有下列语句:

```
int a [20], b [20], f ();
for (i=0, i<20; i++) a [i] =f (i);
out (" fixed array", a); rd (" fixed array"), ? b);
```

rd 操作是成功的, b 与 a 有相同的类型和长度。

虽然这里举的是整型数组, 同样原理可以用于其它类型的数组。

### (3) 不定数组

不定数组域的语法是: array: [length]

这里 array 是一指针, Length 是数组元素的个数。例如, a: 10 表示数组的前面 10 个元素。

例如: m (" array", ? a: 10); 如果配备成功, 则从 Tuple 空间得到数组 a, 长度为 10 的数据。设有如下语句: int \* p, a [20] b [20], c [20], len;

```
p=a;
```

则下列 out 操作在 Tuple 空间有相同的 Tuple 数据:

```
out (" varying array", a: );
out (" varying array", a: 20);
out (" varying array", p: 20);
```

可以用下列语句在 Tuple 空间中存入一个域有 10 个元素的 Tuple:

```
out (" ten elements", a: 10);
out (" ten elements", p: 10);
```

用下列的任何一个 in 操作可得到 10 个元素:

```
out (" ten elements", ? a: len); /* len 将得到结果 */
out (" ten elements", ? p: len); /* len 将得到结果 */
out (" ten elements", ? b: );
out (" ten elements", ? c: );
```

但是用下列的 in 操作不能配备到上面的 out 的 Tuple:

```
out (" ten elements", ? c: );
```

因这里 c 是固定数组, 所以不配备这也是可变数组与固定数组的区别。虽然以上的举例是针对整型数组, 但其它类型的不定数组原理同它一样。

### (4) 多维数组

当多维数组涉及的部件数相同时, 它们配备。考虑如下数组:

```
int a [3] [5] [2], b [4] [6] [2], c [7] [2] [5];
```

下列操作成功:

```
out (" multi", a [0] [0]);
out (" multi", ? b [0] [0]); /* 配备 */
```

而下列操作不成功,

```
out (" multi", a [0]);
out (" multi", ? b [0]);
```

这里 out 操作产生  $5 \times 2 = 10$  个数的部件, 而 in 操作中有  $6 \times 2 = 12$  个数的部件, 两部件所含元素个数不一样, 因而不配备。in 操作用下列操作成功。

```
in (" ten elements", ? c [0]);
```

进行配备数组不要求有相同的维数, 参看下面这些例子:

```
int a [3] [5] [2], b [5] [2], c [2], i;
out (" fixed", a [0]);
in (" fixed", ? b); /* 配备 */
```

```

out (" varying", a [0] [0]);
in (" varying",? c);
/* 配备 */
out (" not an array", a [0] [0] [0]);
in (" not an array",? i); /* 配备 */

```

对于结构类型与结构数组类型的配备原理与以前讨论的相符，这里不再详叙。

#### § 12.1.4 C-Linda 的程序结构、编译、运行

C-Linda 语言与 C 语言比较，除前面介绍的增加了 6 条对 Tuple 空间操作之外，主函数也不相同，在 C 语言中主函数的标记为 main(参数)，而 C-Linda 语言中为 real—main(参数)；除此之外，结构与 C 语言相同。

编译：C-Linda 编译类拟于标准 C 语言编译。C-Linda 的编译程序名为 CLC，而且 C-Linda 的源程序名必须以 CL 为扩展名，如 hello—world. cl；源程序。我们可以用下列命令编译该程序：

```
$ clc -o hello-world hello-world. cl
```

这里-o 选择项表示目标程序名 hello-world，与标准 C 语言一样。

运行：如果运行在网络环境上，编译也应在相同的环境。网络环境下运行时，运行命令为：

```
$ ntsnet 目标文件名 参数
```

分布式环境下运行时，运行命令为：

```
$ ntssent-distribnle 目标文件名 参数
```

#### § 12.2 C-Linda 应用实例

本节举例说明用 C-Linda 语言设计解椭圆型偏微分方程的一种同步并行计算与异步并行计算的方法。

设有单位正方形区域  $\bar{\Omega} = \{0 \leq x, y \leq 1\}$  上椭圆型方程：

$$(0,1) \quad (1,1)$$

$\varphi(x)$	
$\varphi(x)$	$\Delta u = f(x, y)$
	$\varphi(x)$

$$(0,0) \quad (1,0)$$

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \\ u|_{x=0} = \varphi_1(x) \\ u|_{x=1} = \varphi_2(x) \\ u|_{y=0} = \varphi_3(y) \\ u|_{y=1} = \varphi_4(y) \end{cases}$$

把区域  $\bar{\Omega}$  在  $x$  方向和  $y$  方向各分割成  $N$  等份，令  $h = 1/N$ ；则原问题的五点差分格式为：

$$u_{i,j} = [u_{i-1,j} + u_{i+1,j} + u_{i,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}] / 4$$

$$i, j = 1, \dots, N-1$$

用迭代法解上述方程组则得到

$$u_{i,j} = [u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}] / 4$$

$$\varphi_1(jh) = \varphi'_1(i)$$

$$\varphi_2(jh) = \varphi'_2(i)$$

$$\varphi_3(jh) = \varphi'_3(i)$$

$$\varphi_4(jh) = \varphi'_4(i)$$

$$i, j=1, \dots, N-1$$

## (1) 串行算法

用  $u_2(i, j)$  表示  $u_{i,j}^{(k+1)}$ ,  $u_1(i, j)$  表示  $u_{i,j}^{(k)}$ ; 则有内点迭代公式:

$$u_2(i, j) = [u_1(i-1, j) + u_1(i+1, j) + u_1(i, j-1) + u_1(i, j+1) - h^2 f(i, j)] / 4$$

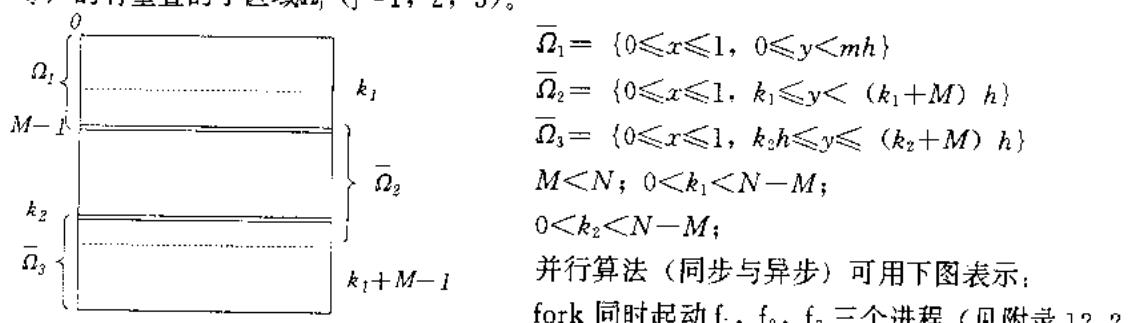
$$i, j=1, \dots, n-1;$$

该问题的串行算法为:

```
begin
    u2, u1 赋初值;
    u2, u1 的边界点赋值;
    while u1=u2;
        内点迭代
        until max |u2(i, j) - u1(i, j)| < ε
    end;
```

## (2) 同步并行算法

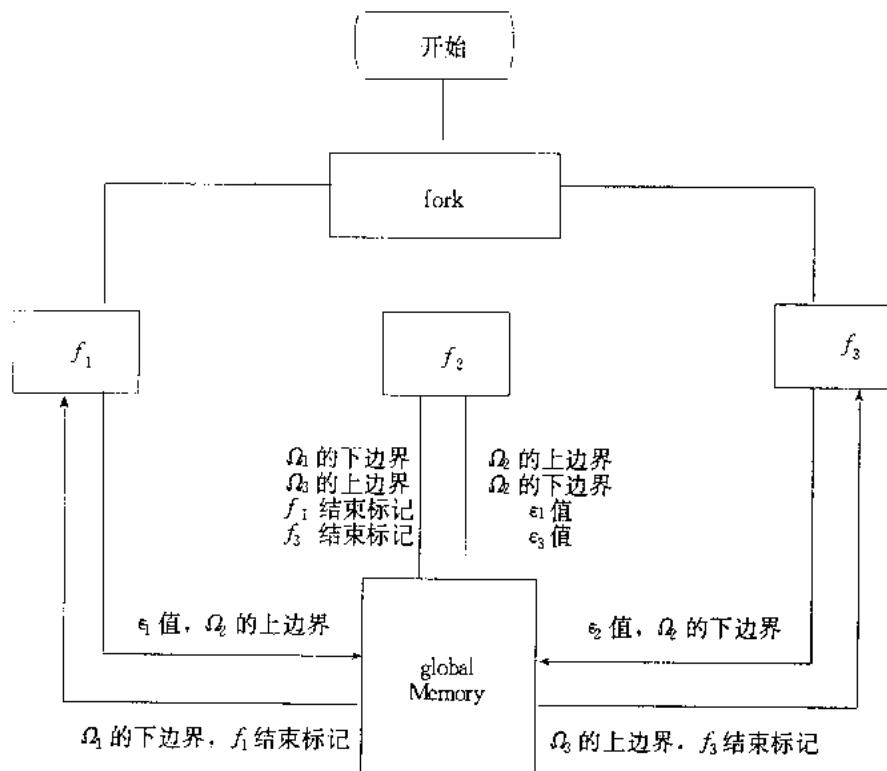
这里用区域分裂法来设计并行程序。将区域  $\bar{\Omega}$  分裂为三个大小相等  $M \times N$  (也可以不等) 的有重叠的子区域  $\bar{\Omega}_j$  ( $j=1, 2, 3$ )。



和附录 12.3 的 real-main() 主函数)。

$f_1$  进程算法 (算法中的  $u_2, u_1$  表示子区域  $\bar{\Omega}_1$  部分的值)。

```
begin
    u2, u1 赋初值;
    u2, u1 左右边界、上边界赋值;
    while u1=u2;
        内点迭代;
        ep1=MAX|u2(i, j) - u1(i, j) |
        输出  $\bar{\Omega}_2$  的上边界值;
        输入  $\bar{\Omega}_1$  的下边界值;
        输出 ep1;
```



```

输入结束标记;
until 结束标记=1
end
f3 进程算法 (算法中的 u2, u1 表示子区域  $\bar{\Omega}_3$  部分的数据)。
begin
u2, u1 赋初值;
u2, u1 左右边界、下边界赋初值;
while u1=u2;
    内点迭代;
        ep3=MAX |u2(i, j) - u1(i, j) |
        输出  $\bar{\Omega}_2$  的下边界值;
        输入  $\bar{\Omega}_3$  的上边界值;
        输出 ep3;
        输入结束标记;
until 结束标记=1
end
f2 进程算法 (算法中的 u2, u1 表示子区域  $\bar{\Omega}_2$  部分的数据)。f2 同时负责 f1, f2, f3 三个
进程迭代结束的判断, 给 f1, f3 发出结束信号。
begin
u2, u1 赋初值;

```

```

u2, u1 左右边界赋值;
while u1=u2;
    内点迭代
        ep2=MAX |u2 (i, j) -u1 (i, j) |
        输出  $\bar{\Omega}_1$  的下边界值;
        输入  $\bar{\Omega}_3$  的上边界值;
        输入  $\bar{\Omega}_2$  的上边界值;
        输入  $\bar{\Omega}_3$  的下边界值;
        输入 ep1;
        输入 ep3;
        根据 ep1, ep2, ep3 判断是否结束, 给结束标记赋值;
        给  $\bar{\Omega}_1$  发出结束标记;
        给  $\bar{\Omega}_3$  发出结束标记;
until 结束标记=1
end

```

$f_1, f_2, f_3$  进程中的输出指往 global memory (Tuple 空间) 写一个 Tuple, 用 out 操作命令; 输出指从 global memory (Tuple 空间) 移出一个 Tuple, 用 in 操作命令。详见附录 12.2。

### (3) 异步并行算法

本节例题从同步并行算法改写成异步并行算法十分简单。首先把同步并行算法中的 in 操作命令改写成 inp 操作函数; 同时, 在往 global memory 写 Tuple 的 out 操作之前用 inp 函数读空该交换数据, 使得 Tuple 空间保持唯一的最新交换数据。详见附录 12.3。

程序说明:

- ① 程序中取  $f(x, y) = 2e^{(x+y)}$ ;  $\varphi_1(x) = e^x$ ;  $\varphi_2(x) = e^{x+1}$ ;  
 $\varphi_3(y) = e^y$ ;  $\varphi_4(y) = e^{y+1}$ ; 该偏微方程的精确解为  $e^{x+y}$ , 对其它用户该类椭圆型方程只改写  $f(x, y)$  函数和边值函数, 该程序也完全实用。
- ② 这些程序全部在 Sun Sparc I 分布式计算机上调试通过。对于格子划分小于 100 时, 并行算法速度并不比串行算法速度快, 对于大于 100 时, 格子划分越小、并行速度比串行算法越快。
- ③ 如果增加内迭代次数, 则并行算法可明显提高速度。
- ④ 在分布式计算机上如果没有其它用户使用时, 异步并行算法与同步并行算法效果基本相同, 如果还有其它用户, 则异步并行算法明显优于同步并行算法。
- ⑤ 附录 12.2 和 12.3 的程序把区域分裂成三块, 使用三台工作站处理器, 对于该类算法的复杂问题用更多台处理器效果更好, 其算法原理完全雷同。

## 附录 12.1 串行程序

```

#define M 111
#define N 111
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int k1, k2;
static float h1=1.0/(N-1);
static float h2=-1.0/(N-1)/ (N-1);
static float e4=0.0001;

float eps (u2, u1)
float u2 [M] [N], u1 [M] [N];
{
    int i, j, q; float ep;
    ep=0.0;
    for (i=1; i < M-1; i++) for (j=1; j < N-1; j++)
        if (fabs (u2 [i] [j] -u1 [i] [j]) >ep) ep=fabs (u2 [i] [j] -u1 [i] [j]);
    return ep;
}
void initu (u2, u1)
float u2 [M] [N], u1 [M] [N];
{
    int i, j;
    for (i=1; i < M-1; i++) for (j=1; j < N-1; j++)
        u2 [i] [j] =u1 [i] [j] =0;
}
void initside (u2, u1)
float u2 [M] [N], u1 [M] [N];
{
    int i, j;
    for (i=0; i < M; i++)
    {
        u2 [i] [0] =u1 [i] [0] =exp (i * h1);
        u2 [i] [M-1] =u1 [i] [M-1] =exp (i * h1+1.0);
        u2 [0] [i] =u1 [0] [i] =exp (i * h1);
        u2 [N-1] [i] =u1 [N-1] [i] =exp (i * h1+1.0);
    }
}

```

```
}

void f1 ( )
{
    float u2 [M] [N], u1 [M] [N], ep=1.0;
    int i, j, k, q;
    initu (u2, u1);
    initside (u2, u1);
    k=0;
    do {
        for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
            u1 [i] [j] =u2 [i] [j];
        k=k+1;
        for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
            u2[i][j]=(u1[i-1][j]+u1[i+1][j]+u1[i][j-1]+u1[i][j+1])
                +2*h2*exp ((i+j)*h1))/4;
        ep=eps (u2, u1);
        printf (" this times is %d eps= %f \n", k, ep);
    }
    while (ep>e4);
}
```

## 附录 12.2 同步并行程序

```
#define M 39
#define N 111
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int k1, k2;
static float e4=0.0001;
static float h2=-1.0/(float)(N-1)*(N-1));
static float h1=1.0/(float)(N-1);

float eps (u2, u1)
float u2 [M] [N], u1 [M] [N];
{
    int i, j; float ep, epl;
    ep=0.0;
    for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
    { epl=fabs (u2 [i] [j] -u1 [i] [j]);
        if (epl>ep) ep=epl;
    }
    return ep;
}

void initu (u2, u1)
float u2 [M] [N] u1 [M] [N];
{
    int i, j;
    for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
        u2 [i] [j] =u1 [i] [j] =0;
}

void initside (u2, u1, k)
float u2 [M] [N], u1 [M] [N];
int k;
{
    int i, j;
    for (i=0; i<M; i++)
    {
        u2 [i] [0] =u1 [i] [0] =exp (h1 * (i+k));
    }
}
```

```

    u2 [i] [N-1] =u1 [i] [N-1] =exp (h1 * (i+k) +1.0);
}
for (j=0; j<N; j++)
{
    u2 [0] [j] = u1 [0] [j] = exp (h1 * (float) (j+k));
    u2 [M-1] [j] =u1 [M-1] [j] =exp (h1 * (float) (j+k+M-1));
}
}

f1 (outline, beginline)
int outline, beginline;
{
    float u2 [M] [N], u1 [M] [N], ep=1.0;
    int i, j, k, q;
    initu (u2, u1);
    initside (u2, u1, beginline);
    k=0;
    do {
        for (i=0; i<M; i++) for (j=0; j<N; j++)
            u1 [i] [j] =u2 [i] [j];
        k=k+1;
        for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
            u2[i][j]=(u1[i-1][j]+u1[i+1][j]+u1[i][j-1]+u1[i][j+1])
                +2 * h2 * exp ((i+j) * h1)) /4;
        ep=eps (u2, u1);
        out (" block11", u2 [outline]);
        in (" block21",? u1 [M-1]);
        out (" eps1", ep);
        in (" endl",? q);
    }
    while (q!=1);
/*
    for (i=0; i < M; i++) {
        printf ("%3d", i+beginline);
        for (j=0; j<N; j++) printf ("%5.3f", u1 [i] [j]);
        printf ("\n"); */
}
f3 (outline, beginline)
int outline, beginline;
{

```

```
float u2 [M] [N], u1 [M] [N], ep=1.0;
int i, j, k, q;
initu (u2, u1);
initside (u2, u1, beginline);
k=0;
do {
    for (i=0; i<M; i++) for (j=0; j<N; j++)
        u1 [i] [j] =u2 [i] [j];
    k=k+1;
    for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
        u2[i][j]=(u1[i-1][j]+u1[i+1][j]+u1[i][j-1]+u1[i][j+1])
            +2 * h2 * exp ( (i+j) * h1) /4;
    ep=eps (u2, u1);
    out (" block33", u2 [outline]);
    in (" block22",? u1 [0]);
    out (" eps3, ep");
    in (" end3",? q);
}
while (q!=1);
/*
for (i=0; i<M; i++) {
    printf ("%3d", i+beginline);
    for (j=0; j<N; j++) printf ("%5.3f", u1 [i] [j]);
    printf ("\n");} */
void f2 (outline1, outline2, beginline)
int outline1, outline2, beginline,
{
    float u2 [M] [N], u1 [M] [N], ep1, ep3, ep2=1.0;
    int i, j, k, q;
    initu (u2, u1);
    initside (u2, u1, beginline);
    k=0;
    do {
        for (i=0; i<M; i++) for (j=0; j<N; j++)
            u1 [i] [j] =u2 [i] [j];
        k=k+1;
        for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
            u2 [i] [j]=(u1 [i-1] [j] +u1 [i+1] [j] +u1 [i] [j-1] +u1 [i] [j+1])
```

```

+2 * h2 * exp ( (i+j) * h1)) /4;
ep2=eps (u2, u1);
out (" block21", u2 [outline1]);
out (" block22", u2 [outline2]);
in (" block11",? u1 [0]);
in (" block33",? u1 [M-1]);
in (" eps1",? ep1);
in (" eps3",? ep3);
if ( (ep1<e4) && (ep2 <e4) && (ep3<e4))
q=1;
else
q=0;
out (" end1", q);
out (" end3", q);
printf (" this times is %d eps=%f %f %f \n",
k, ep1, ep2, ep3);
}
while (q!=1);
/*
for (i=0; i<M; i++) {
printf ("%3d", i+beginline);
for (j=0; j<N; j++) printf ("%5. 3f", u1 [i] [j]);
printf ("\n");} */
}
real_main () /* fork */
{
eval (" hua", f1 (M-3, 0));
eval (" hua", f3 (2, 12));
f2 (2, 6, 6);
in (" hua",? int);
in (" hua",? int);
}

```

### 附录 12.3 异步并行程序

```
#define M 19
#define N 51
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int k1, k2;
static float e4=0.0001;
static float h2=-1.0/(float)(N-1)*(N-1));
static float h1=1.0/(float)(N-1);

float eps (u2, u1)
float u2 [M] [N], u1 [M] [N];
{
    int i, j; float ep, epl;
    ep=0.0;
    for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
        { epl=fabs (u2 [i] [j] -u1 [i] [j]);
        if (epl>ep) ep=epl;
    }
    return ep;
}
void initu (u2, u1)
float u2 [M] [N], u1 [M] [N];
{
    int i, j;
    for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
        u2 [i] [j] =u1 [i] [j] =0;
}
void initinside (u2, u1, k)
float u2 [M] [N], u1 [M] [N];
int k;
{
    int i, j;
    for (i=0; i<M; i++)
```

```

{
    u2 [i] [0] = u1 [i] [0] = exp (h1 * (i+k));
    u2 [i] [N-1] = u1 [i] [N-1] = exp (h1 * (i+k) +1.0);
}

for (j=0; j<N; j++)
{
    u2 [0] [j] = u1 [0] [j] = exp (h1 * (float) (j+k));
    u2 [M-1] [j] = u1 [M-1] [j] = exp (h1 * (float) (j+k+M-1));
}
}

f1 (outline, beginline) .
int outline, beginline;
{
    float u2 [M] [N], u1 [M] [N], u [N], ep=1.0;
    int i, j, k, q;
    initu (u2, u1);
    initside (u2, u1, beginline);
    k=0;
    do {
        for (i=0; i<M; i++) for (j=0; j<N; j++)
            u1 [i] [j] = u2 [i] [j];
        k=k+1;
        for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
            u2[i][j]=(u1[i-1][j]+u1[i+1][j]+u1[i][j-1]+u1[i][j+1]
                      +2*h2*exp ((i+j)*h1))/4;
        ep=eps (u2, u1);
        while (inp (" block11",? u) ==1);
        out (" block11", u2 [out|ine]);
        inp (" block21",? u1 [M-1]);
        while (inp (" eps1",? int) ==1);
        out (" eps1", ep);
        inp (" endl",? q);
    }
    while (q!=1);
    /*
    for (i=0; i<M; i++) {
        printf ("%3d", i+beginline);
    }
}

```

```

    for (j=0; j<N; j++) printf ("%5. 3f", u1 [i] [j]);
    printf ("\n");} /* */
}

f3 (outline, beginline)
int outline, beginline;
{
    float u2 [M] [N], u1 [M] [N], u [N], ep=1.0;
    int i, j, k, q;
    initu (u2, u1);
    initinside (u2, u1, beginline);
    k=0;
    do {
        for (i=0; i<M; i++) for (j=0; j<N; j++)
            u1 [i] [j] =u2 [i] [j];
        k=k+1;
        for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
            u2[i][j]=(u1[i-1][j]+u1[i+1][j]+u1[i][j-1]+u1[i][j+1])
                +2*h2 * exp ((i+j) * h1)) /4;
        ep=eps (u2, u1);
        while (inp (" block33",? u) ==1);
        out (" block33", u2 [outline]);
        inp (" block22",? u1 [0]);
        while (inp (" eps3",? int) ==1);
        out (" eps3", ep);
        inp (" end3",? q);
    }
    while (q!=1);
/* */
    for (i=0; i<M; i++) {
        printf ("%3d", i+beginline);
        for (j=0; j<N; j++) printf ("%5. 3f", u1 [i] [j]);
        printf ("\n");} /* */
}

void f2 (outline1, outline2, beginline)
int outline1, outline2, beginline;
{
    float u2 [M] [N], u1 [M] [N], u [N], ep1, ep3, ep2=1.0;

```

```

int i, j, k, q;
initu (u2, u1);
initinside (u2, u1, beginline);
k=0;
do {
    for (i=0; i<M; i++) for (j=0; j<N; j++)
        u1 [i] [j] =u2 [i] [j];
    k=k+1;
    for (i=1; i<M-1; i++) for (j=1; j<N-1; j++)
        u2[i][j]=(u1[i-1][j]+u1[i+1][j]+u1[i][j-1]+u1[i][j+1]
                  +2 * h2 * exp ((i+j) * h1)) /4;
    ep2=eps (u2, u1);
    while (inp (" block21",? u) ==1);
    out (" block21", u2 [outline1]);
    while (inp (" block22",? u) ==1);
    out (" block22", u2 [outline2]);
    inp (" block11",? u1 [0]);
    inp (" block33",? u1 [M-1]);
    inp (" eps1",? ep1);
    inp (" eps3",? ep3);
    if ((ep1<e4) && (ep2<e4) && (ep3<e4))
        q=1;
    else
        q=0;
    while (inp (" epsl",? int) ==1);
    out (" endl", q);
    while (inp (" epsl",? int) ==1);
    out (" end3", q);
    printf (" this times is %d eps=%f %f %f \n",
           k, ep1, ep2, ep3);
}
while (q!=1);
/*
for (i=0; i<M; i++) {
    printf ("%3d", i+beginline);
    for (j=0; j<N; j++) printf ("%5. 3f", u1 [i] [j]);
    printf ("\n");} */

```

```
}

real_main () /* fork */

{
    eval (" hua", f1 (M-3, 0));
    eval (" hua", f3 (2, 12));
    f2 (2, 6, 6)
        in (" hua",? int);
    in (" hua",? int);
}
```

## 参考文献

- 1 康立山, 孙乐林, 陈毓屏. 解数学物理问题的异步并行算法. 北京: 科学出版社, 1985
- 2 康立山, 全惠云. 数值解高维偏微分方程和分裂法. 上海: 上海科学出版社, 1990年
- 3 康立山, 陈毓屏. 并行算法简介(上). 数值计算与计算机应用, Vol 9, No. 3 (1988), 169~177
- 4 康立山, 陈毓屏. 并行算法简介(下). 数值计算与计算机应用, Vol 9, No. 4 (1988), 245~252
- 5 康立山, 陈毓屏. 并行计算发展及异步并行算法. 并行算法学术会议论文集(1988), 1 ~4
- 6 康立山. Schwarz 交替法的推广. 武汉大学学报(自然科学版) 1979年, 第四期, 11~23
- 7 康立山, 陈毓屏, 邱毓兰. Schwarz 算法及其在分布式并行处理机上的应用(I)—解弱非线性椭圆型边值问题. 见: 分布式并行处理系统探索. 武汉: 武汉大学出版社. (1984), 115~122
- 8 康立山, 陈毓屏. Schwarz 算法及其在分布式并行处理机上的应用(II)—解定常二维 Navier-Stokes 方程. 见: 分布式并行处理系统探索. 武汉: 武汉大学出版社, (1984), 80~91
- 9 康立山. 解数学物理问题的异步并行算法. 数学物理学报, 1983 (3): 483~494
- 10 康立山, 邱毓兰, 陈毓屏, 彭德纯. Schwarz 算法及其在分布式并行处理机上的应用(I)—解线性椭圆型边值问题. 见: 分布式并行处理系统探索. 武汉: 武汉大学出版社, 1984, 71~79
- 11 康立山, 陈毓屏, 孙乐林. Schwarz 算法及其在分布式并行处理机上的应用(V)—解非定常数学物理问题. 见: 分布式并行处理系统探索. 武汉: 武汉大学出版社, 1984, 92~101
- 12 陈毓屏, 康立山. Schwarz 算法及其在分布式并行处理机上的应用(N)—解奇异性椭圆型边值问题. 见: 分布式并行处理系统探索. 武汉: 武汉大学出版社, 1984, 102~114
- 13 Kang Lishan, Chen Yuping, Sun Lelin and Quan Huiyun. The asynchronous parallel algorithms S-COR for solving P. D. E.'s on multiprocessors, International Journal of Computer Mathematics, Vol. 1, (1985), 163—172
- 14 Kang Lishan, Evans, D. J., The convergence rate of the Schwarz alternating procedure (I)—For Neumann problems, International Journal of Computer Mathematics, Vol. 21, (1987), 85~108

- 15 Kang Lishan. Parallel algorithms and domain decomposition, Lecture Notes in Mathematics, 1297. Springer—Verlag, (1987), 61~75
- 16 Kang Lishan, Evans, D. J., The convergence rate of the Schwarz alternating procedure (V) —For more than two subdomains, Inter. J. Computer Math., Vol. 23, (1988), 295~313
- 17 Kang Lishan, Evans, D. J., Strategies of domain decomposition for designing parallel algorithms, Proceedings of Parallel Computing 89, (1989).
- 18 Kang Lishan (ed.), Parallel Algorithms and Domain Decomposition, Wuhan: University Press, (1987).
- 19 Lishan Kang and Yunin Chen, Asynchronous parallel algorithms based on DDM for

- tran*, Scientific Programming 1, 1, August 1992, Also published as: ACPC/TR 92-3, Austrian Center of Parallel Computation, March 1992.
- 34 Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, ChauWen Tseng, and Min-You Wu. *Fortran D Language Specification*. Report COMP TR90-141 (Rice) and SCCS-42c (Syracuse), Department of Computer Science, Rice University, Houston, Texas, and Syracuse Center for Computational Science, Syracuse University, Syracuse, New York, April 1991.
- 35 High Performance Fortran Forum. *High Performance Fortran Language Specification* Scientific Programming, 2, 1, 1993. Also published as: CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised May. 1993). Also published as: Fortran Forum, 12, 4, Dec. 1993 and 13, 2 June 1994.
- 36 High Performance Fortran Forum. *High Performance Fortran Language Specification, version 1.0* CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised May. 1993).
- 37 C. Koelbel and D. Loveman and R. Schreiber and G. Steele, Jr. and M. Zosel. *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994.
- 38 C. Koelbel and P. Mehrotra. An Overview of High Performance Fortran. *Fortran Forum*, Vol. 11, No. 4, December, 1992.
- 39 David B. Loveman. High Performance Fortran. IEEE Parallel 43 Distributed Technology, Vol. 1, No. 1, February 1993.
- 40 David B. Loveman. Element Array Assignment-the FORALL Statement. Third Workshop on Compilers for Parallel Computers, Vienna, Austria, July 6-9, 1992.
- 41 MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California. MasPar Fortran Reference Manual, May 1991. [Software Version 1.1, 9303-0000, Rev. A2].
- 42 Piyush Mehrotra and J. Van Rosendale. Programming Distributed Memory Architectures Using Kali. In: Nicolau, A. et al. (Eds); *Advances in Languages and Compilers for Parallel Processing*, pp. 364-384, Pitman/MIT-Press, 1991.
- 43 Andrew Meltzer, Douglas M. Pase, and Tom MacDonald. *Basic Features of the MPP Fortran Programming Model*, Cray Research, Inc, Eagan, Minnesota, August 19, 1992.
- 44 John Merlin. Techniques for the Automatic Parallelisation of ‘Distributed Fortran 90’, Technical Report SNARC 92-02, Dept. of Electronics and Comp. Science, Univ. of Southampton, November 1991.
- 45 Robert E. Millstein. Control Structures in ILLIAC IV Fortran. Communications of the ACM, 16 (10) : 621-627, October 1973.
- 46 Douglas M. Pasa, Tom MacDonald, and Andrew Meltzer. MPP Fortran Programming Model, Cray Research, Inc, Eagan, Minnesota, August 26, 1992.

- 
- 47 Guy L. Steele Jr. High Performance Fortran: Status Report. in Workshop on Languages, Compilers, and Runtime Environments for Distributed-Memory Multiprocessors, *ACM SIGPlan Notices*, Vol. 28, No. 1, January 1993.
  - 48 US Department of Defense. Military Standard, MIL-STD-1753: FORTRAN, DoD Supplement to American National Standard X3. 9-1978, November 9, 1978.
  - 49 Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran -a Language Specification, ICASE Interim Report 21, ICASE NASA Langley Research Center, Hampton, Virginia 23665, March 1992.
  - 50 Kang Lishan, Iain Macleod, Chen Lijuan, Chen Yuping. Asynchronous parallel algorithms based on DDM on CM-5 (I); for solving linear elliptic PDE's, 1995.
  - 51 Quan Huiyun, Sun Lelin, Gao Hanping, Asynchronous parallel algorithms based on DDM for solving some nonlinear PDE's. Wuhan University Journal of Natural Sciences, 1996.
  - 52 Daniel H. L. CM. Fortran Language reference manual. Massachusetts: Thinking Machines corporation, 1994
  - 53 严掘非, 刘尚德译. 大规模并行计算机—Connection Machine. 南京: 南京大学出版社, 1992
  - 54 [美] 黄铠著. 高等计算机系统结构. 王鼎兴等译. 北京: 清华大学出版社与广西科学技术出版社联合出版, 1995.