基于规范划分集的并行循环编译框架*)

A Compiling Framework Based on Uniform Partitioning Schemes for Parallel Loops

黄其军 丁 阳 余华山 丁文魁 许卓群
(北京大学计算机科学技术系 北京 100871)

Abstract Compilation of parallel loops is one of the most important parts in parallel compilation and optimization. This paper mainly discusses the key techniques during the compilation implementation of parallel loops, based on the uniform partition schemes. It includes techniques in local array index generating, loop space reconstructing, communication detecting and organizing and data dependence disposing. The efficiency of this implementation has been proved by lots of experiments. The p_HPF compiler which adopts this compiling framework can obtain good speedups and efficiencies. The compiler has been applied in many fields, particularly the field of petroleum exploration.

Keywords Parallel loop, Parallel compilation, Computation partition, Parallel computation, Communication disposing, Data dependence disposing

1 引言

并行循环描述大多数先进科学应用的核心部分,是颇有价值的并行性来源[1]。高级并行语言往往提供专门指导语句表达并行循环,以便并行编译器利用并行循环的非数据相关特性。在数据并行程序设计语言 HPF 中规定,可以将关键词 INDEPENDENT 加在 Do 循环前面,指出其后的 Do 循环是一个并行循环。与之相关的编译实现也成为 HPF 编译器的一个重点。

并行循环 L 可以严格定义如下:假设 R(i)表示迭代 i 所引用的变量集合,W(i)表示迭代 i 所更新的变量集合;如果对循环 L 的任意两次迭代 i_1 和 i_2($i_1 \neq i_2$),满足:

$R(i_1) \cap W(i_2) = \phi$ 且 $W(i_1) \cap R(i_2) = \phi$ 且 $W(i_1) \cap W(i_2) = \phi$

则称 L 为并行循环。n(n≥1)个具有不同循环控制变量的并行循环可以通过完全嵌套(完全嵌套指内层循环是外层循环内唯一语句的循环嵌套形式)组成 n 重并行循环,在下文中,n(n≥1)重并行循环统称并行循环。

对分布内存多处理器体系结构,如果把并行循环迭代尽可能均匀地分配到不同处理器执行,那么循环内的并行性就可开发出来。计算分配到处理器后,各个处理器在进行计算时往往需要访问其他处理器内存中的数据,需要通过通信来完成;通信开销的大小从另一个侧面反映计算分配的合理性。计算分配工作叫做计算划分,计算划分既要把计算量在处理器间均匀分配,达到负载均衡,又要尽量减少通信开销。需要通过选择最优的计算划分方案来实现,不同计算划分方案的负载均衡性不同,带来的通信开销也不同。

计算划分之后,应组织必要的通信语句完成非本地数据的访问。高效地组织通信语句、减少通信代价,对获得高性能必不可少。因此通信及其优化是实现并行循环的另一个重要问题。

我们开发的 p_HPF 编译器在并行循环语句编译处理方面具有很高的性能,在实现策略方面也独具特色,以规范划分集计算划分[2]为基础,根据计算需求实现通信外提和群通信,同时妥善地处理了数据相关等问题,最终生成结点程序。

同时,在国际上具科学研究性质的 HPF 编译器中,有代表性的还有 RICE 大学的 dHPF,Stanford 大学的 SUIF 等;商业 HPF 编译器有 IBM 的 pHPF,PGI 的 pgHPF 等。这些编译器在并行循环处理方面各有特色,与我们的编译器相比在计算划分策略以及涉及到的并行粒度、负载均衡评估方面各不相同,同时计算划分之后的通信及其优化方面也有差异。另外,国际并行研究领域从理论上提出幺模变换算法[3,4],数据和计算分解加障碍消除算法[4]以及仿射计算划分算法[5,6]等循环并行化方法。在与这些编译器的实现进行比较之后,我们的编译处理方法显得更加简单而有效,并且已经应用于石油勘探和化学分子模拟等,并取得良好效果。

本文首先介绍数据并行编译的若干问题,指出并行循环语句编译的重点和难点;接着论述 p_HPF 关于并行循环语句的编译实现技术,包括计算划分、局部下标生成和循环空间重构、通信检测和通信布置以及相关问题处理等。其中计算划分采用基于规范划分集的计算划分算法,在文[2]中有详细论述,本文重点论述其他几方面,并与国际最新研究成果进行了比较;最后给出测试结果,使用 FFT 程序和 SPEC92 的 nasa7 基准测试程序检验了编译处理的效果,并指出进一步工作设想。

2. 并行循环语句编译的若干问题

数据并行程序的特点是对大量数据执行相同操作,这使得它很适合采用 SPMD 的结点程序。p_HPF 编译器就是采用这种方式,将 HPF 源程序编译变换成 f77+运行支持库的 SPMD 程序,再使用平台本地编译器将结点程序转换成可执行程序。

2.1 并行循环语句编译的几个关键问题

HPF 语言提供 ALIGN 和 DISTRIBUTE 指导语句,以便程序员描述数据在分布内存多处理器间的分布,这个过程就是数据划分,如例 1 所示。数据分布到 p 个处理器将产生 p 个不连续的数组片段,每个处理器只能访问分配到本地的数据元素。因此,编译器必须产生在每个处理器上执行的代码(结点程序)直接访问本地数组片段,同时插入通信进行非本地数据访问。

例 1:

```
REAL A(200),B(200)
! hpf$ template T(200)
! hpf$ processors procs(4)
! hpf$ align with T :: A,B
! hpf$ distribute T(cyclic(25)) onto procs
! hpf$ independent
```

*)本项研究由自然科学基金(编号 60173004)资助。

```
Do I=2,200
  A(I) = B(I-1)
End do
```

例1 程序对应的结点程序

```
    REAL A(200),B(200)
my $p = myproc() { *返回0…3 * }
Do I=2,200
  If(my $p .eq. owner(B(I-1))) then
    Send B(I-1) to owner(A(I))                    ①
  Endif
  If(my $p .eq. owner(A(I))) then
    Receive B(I-1) from owner(B(I-1))             ②
    A(I) = B(I-1)
Endif
End do
```

在例1结点程序中，按 HPF 程序中数组大小声明了数组 A 和 B，循环空间保持不变。结点程序并行计算思路是：循环内首先由拥有 B(I-1)的处理器将 B(I-1)发送给 A(I)的拥有者，不拥有 B(I-1)的处理器跳过这一步；计算由 A(I)的拥有者负责，它首先接受 B(I-1)，然后进行计算，其它处理器跳过。由这个小程序的编译设想，我们看到编译器首先面临的几个问题：

1. 计算划分　例1结点程序把循环迭代中的计算(一条赋值语句)安排在拥有赋值左部的处理器上进行，这实际上是一种简单的计算划分策略，称为拥有者计算原则。在一般的并行循环语句中，循环内计算部分有多条语句，语句类型也很丰富；而且计算中所涉及的数组具有各种不同的分布方式。因此对并行循环语句来说，不能简单使用拥有者计算原则来进行计算划分，我们开发了基于规范划分集的计算划分算法[2]来完成这个工作。

2. 全局到局部的转换　将 HPF 程序中全局数组访问转变成结点程序中局部数组访问。数据划分将全局数组对应到局部数组，程序计算中数据访问形式也要相应变换；全局数组一般与局部数组同名，因此访问形式变换主要是数组下标转换；也就是把 HPF 程序中数组的全局下标转变成结点程序中的局部下标。

3. 通信语句插入　结点程序只能访问本地数据，对于非本地数据要靠通信来访问。通信时，数据访问者接收数据，数据拥有者发送数据；各处理器相互协同完成整个计算任务。因此，编译器要在结点程序中插入接收和发送等通信语句，并进行整体的通信规划。如例1中语句①、②所示。

进一步考虑，例1中的结点程序不可能有较好的效率，首先所有处理器执行整个循环空间，存在大量冗余迭代和条件判断；其次开销很大的通信语句放在循环内，造成并行程序效率严重降低。为了提高结点程序的效率，编译器还要进行如下工作：

4. 循环空间重构和冗余迭代消除　将循环的全局循环空间替换为局部循环空间，不但可以适应结点程序对局部数据的访问，而且可以消除冗余迭代，提高结点程序运行效率。

5. 通信语句外提和群通信[7]　将循环内的通信语句提到循环外进行，避免费时的通信过程在循环内引起程序效率严重降低。在例1结点程序中，通信是对一次迭代涉及的单个数据元素进行的，通信语句外提后，可以把整个循环要访问的某个非本地数组通过一次群通信[7]完成接收和发送。这样做的好处在于：一方面，把零散通信数据集中发送和接收，会增大通信数据包，减少通信信道访问次数，提高效率；另一方面，群通信屏蔽掉了各结点之间数目众多的点到点通信的实现细节，可以简化结点程序生成过程。另外，可以针对群通信中的某些通信模式进行仔细优化，进一步提高效率。

6. 通信检测和通信布置　在进行群通信时，根据通信数组的分布情况，可以把通信模式分为无通信，SHIFT 通信和 REMAP 通信[7]。通信检测确定具体通信模式，通信布置就是按通信模式在结点程序中安排相应通信语句。

7. 相关问题处理　再进一步考虑，上面的 HPF 例子只是一个相当简单的并行循环语句。而在一般并行循环语句中，存在多条不同类型的语句，以及不同语句对同一数组的访问；这就需要对同一数组的不同访问进行相关处理，例如下面一个的并行循环语句：

例2 迭代内存在相关性的并行循环语句

```
!HPF $ INDEPENDENT
       DO I=2,99
!HPF $ INDEPENDENT
       DO J=3,99
          A(I, J)=B(I+1, J-2)                     ①
          D(I-1, J-1)=E(J, I)+ F(A(I,J), c)       ②
       END DO
```

这里语句①和②都访问了数组 A，结点程序中必须使②正确访问①已更新的有关数据。

2.2 并行循环语句编译问题小结

在2.1节，我们给出并行循环语句编译的关键问题，按逐渐深入的思路提出了7点值得注意的地方。在较好地解决了计算划分问题的基础上，后续处理问题成为影响编译器性能的重要因素，下面我们在简要介绍 p_HPF 中并行循环编译处理过程之后，重点论述其中局部下标生成和循环空间重构、通信检测和通信布置以及相关问题处理的解决方案。

3. p_HPF 关于并行循环语句编译的技术

图1显示了 p_HPF 编译器中处理并行循环的流程，下面简要叙述处理过程中的关键部分。

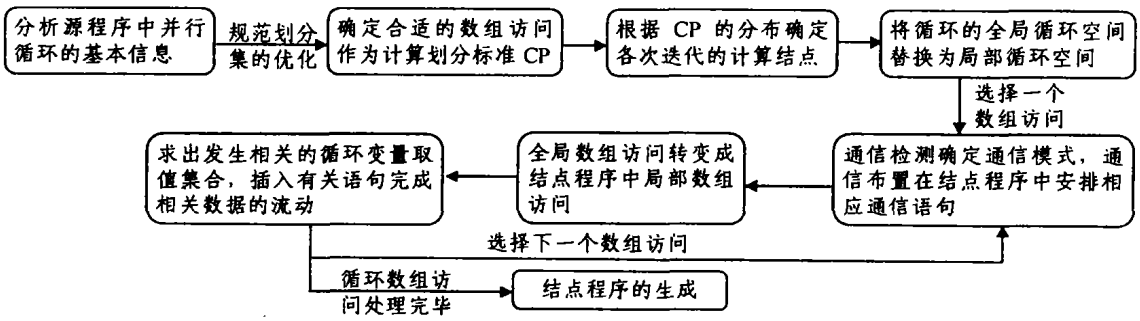

图1 p_HPF 编译器处理并行循环的过程

3.1 计算划分

国际上出现的 HPF 编译器在并行循环处理的计算划分方面采用了不同的策略，可分为三类：一类是让循环内各个语句拥有各自的计算划分方案，各语句的计算划分方案共同构

成整个循环的计算划分方案;Rice 大学的 dHPF 属于这一类,这是在较细粒度上开发并行性,意味着可能获得更大程度的并行;但其缺点在于无法评估负载均衡性(dHPF 不进行负载平衡)(http://www.cs.rice.edu/~dsystem/dhph/dhpf-overview-96/index.html)。另一类是循环内各语句拥有统一的计算划分方案,循环迭代被作为整体来分配,并行粒度大;我们提出的基于规范划分集的计算划分算法[2]和 Stanford 大学的 SUIF 中使用的算法属于第二类,虽然在计算划分方案上丧失了某些一般性,但迭代被作为整体来分配,编译器可以通过比较各个结点所分配的迭代数来评估负载均衡性,进行负载平衡。第三类是拥有者计算原则,即根据数据元素的赋值语句,数据元素的新值由拥有该数据元素的处理器负责计算。这是一类简单策略,仅仅适用于相对整齐的循环,但一般说来优化性能不好。

我们所采用的基于规范划分集的计算划分算法[2]主要思路是:

在并行循环的循环体中,通过规范划分集的优化,选取确定某个合适的数组访问作为计算划分标准 CP;以 CP 的数据分布确定循环迭代的分布,各次迭代的计算结点就是该迭代所访问的 CP 数组元素所在结点;在结点间进行计算分配时,循环迭代被当作不可分割的整体。在选取 CP 时,我们综合考虑通信和负载均衡这两个影响并行效率的重要因素,力求使并行循环在最短时间内完成。

3.2 局部下标生成和循环空间重构

3.2.1 *HPF 数据映射下的规则下标序列* 我们把可以表示为 A(l:h:s)形式的数组下标序列称为规则下标序列,称这样的数组访问模式为规则访问模式,规则访问模式是并行循环语句中的主要数组访问模式。研究规则下标序列的特性对并行编译处理特别是数组局部下标生成和循环空间重构是很重要的。在 HPF 的数据映射机制下,一个全局的规则下标序列将被映射到每个结点上的局部下标序列,映射后的局部下标序列并不一定保持规则,我们可以看看一个简单的例子来说明这一点。

数组 A 的映射:

```
!hpf$ template T(100)
!hpf$ processors procs(3)
!hpf$ align A with T
!hpf$ distribute T(cyclic(4)) onto procs
```

如果访问 A(1:100:5),所涉及的数组元素的分布情况和局部下标可由图2表示。

Processor 0

0^{0}	1	2	3
12	13	14	15^{7}
24	25^{9}	26	27
36	37	38	39
48	49	50^{18}	51
60^{20}	61	62	63
.....			

Processor 1

4	5^{1}	6	7
16	17	18	19
28	29	30^{10}	31
40^{12}	41	42	43
52	53	54	55^{19}
64	65^{21}	66	67
.....			

Processor 2

8	9	10^{2}	11
20^{4}	21	22	23
32	33	34	35^{11}
44	45^{13}	46	47
56	57	58	59
68	69	70^{22}	71
.....			

图2 全局规则下标序列在局部表现为不规则

图中每个处理器上深色矩形框内数组元素是访问 A(1:100:5)所涉及的数据,框中前面数字表示该元素全局下标,后面用上标形式给出它的局部下标。可以看出虽然访问的全局下标序列是规则的,但局部下标不规则,比如 Processor 1上的局部下标序列为(1,10,12,19,21,…)。

对于上述情况,如何在结点程序进行局部数组访问将是一个复杂问题,国际数据并行编译研究领域提出了一些解决办法,比如基于格子(lattice)的地址生成技术[8]等。

在应用领域最为常用的分布方式是 BLOCK 和 CYCLIC 方式,我们针对这两种分布方式研究发现,在 HPF 两级映射中,如果采用 BLOCK 或 CYCLIC 方式对模板进行分布,那么全局规则数组访问映射到处理器上的局部访问也是规则的。

HPF 数据映射的一般形式可表示为:

```
type_def A(L:U)
!hpf$ template T(Lt:Ut)
!hpf$ processors procs(num)
!hpf$ align A(i) with T(a*i+b)
!hpf$ distribute T(dist_type) onto procs
```

即如果数据分布类型 dist_type=BLOCK 或 dist_type=CYCLIC,任意数组片段 A(l:u:s)散布于任何处理器上的局部数组片段可表示为 A(l_l:l_u:l_s),其中 l,u,s 为全局下界、上界和步长,l_l,l_u,l_s 为局部下界、上界和步长,处理器个数为 num,则每个处理器上的局部数组元素相应全局下标序列也是规则的,可表示为(x:y:z)的形式。

容易得出,这一结论推广到多维也是成立的。这样就保证了 p_HPF 编译器的局部下标生成技术和循环空间重构技术的正确性。这两个技术分别在下文3.2.2节和3.2.3节叙述。

3.2.2 *p_HPF 编译器中局部下标生成技术* HPF 程序采用全局命名空间,直接访问全局数组的任何一个元素,这是一种全局访问模式;而编译后的结点程序只能直接访问本地数据,是局部访问模式。因此编译器必须解决全局访问模式到局部访问模式的转换问题,将数组的全局下标变换成局部下标。一般说来,对于 HPF 程序中一个普通的数组访问 A(l:u:s),尽管它的形式是规则的,但在 HPF 的数据映射机制下,映射到结点的局部数组访问可能不规则(见图2),这时局部下标序列一般要放到一个表中,结点程序中的局部下标需要借助中间数组确定。但由于 HPF 常用数据映射方式下,全局规则数组片段映射到结点的局部数组片段仍然保持规则,只要计算出局部数组片段的下标下界、上界和步长,就可以用简单表达式表示出局部下标,确定数组局部访问模式。

具体地说,数组访问模式从全局到局部的转换中我们采用一种简洁的方式:在结点程序中,首先调用一个运行支持函数,该函数根据数组的分布数组描述子(DAD)和所要访问的全局下标序列的下界 l、上界 u 和步长 s,以及本处理器号,计算出本结点相应局部下标序列,用下界 l_l、上界 l_u 和步长 l_s 来确定;然后利用 l_l、l_u 和 l_s 表示出每次迭代所要访问的数组元素的局部下标。如图3所示。

…

```
!HPF $ INDEPENDENT
  DO I=i_l, i_u, i_s
      ...
      access(A(l : u : s))
      ...
  END DO
...
---------------
...
CALL local_addr(DAD_A,l,u,s,mp,l_l,l_u,l_s)
do I=li_l, li_u, li_s
   ...
   access(A(l_l: l_u: l_s)
   ...
end do
    ...
```

图3　局部下标生成法

```
!HPF $ INDEPENDENT
  DO I=i_l,i_u,i_s
      ...
      access(CP(a*I+b))
      ...
  END DO
...
------------
...
CALL local_iter(DAD_CP,i_l,i_u,i_s,m_p,a,b,li_l,li_u,li_s)
do I=li_l,li_u,li_s
   ...
   access(CP(local_address)
...
end do
...
```

图4　循环空间重构法

图3虚线上面部分是一个一般形式的并行循环语句，其中访问了全局数组片段 A(l:u:s)。为了说明局部下标生成方法，虚线下面部分给出结点程序，其中 local_addr 是编译器提供的运行支持函数，用来计算局部下标的下界、上界和步长。参数 DAD_A 是数组 A 的 DAD，l、u 和 s 是要访问的全局下标下界、上界和步长，mp 是本处理器的 ID，该函数根据这些参数计算出本地局部下标序列的下界、上界和步长，通过参数 l_l、l_u 和 l_s 传回，循环中数组访问形式相应变换为局部访问模式 $A(l_l:l_u:_s)$。

当然，上面假设数组访问 A(l:u:s)无通信，如果 A 需要通信，局部数组访问仍然可以表示为$(l_l:l_u:l_s)$的形式。事实上，对这种情况，我们将访问所需数据通过通信语句映射到本地一个临时空间中，并使数据在临时空间按计算划分标准 CP 的局部数据排列方式排列，由3.2.1节结论临时空间中的数据也是规则排列的，这时在结点程序中对数组 A 的访问形式可表示为 $TMP\_A(l_l:l_u:l_s)$，TMP_A 表示临时空间。

3.2.3　*p_HPF 编译器中循环空间重构技术*　图3上面部分并行循环的全局计算空间$(i_l:i_u:i_s)$在结点程序中被变换为局部计算空间$(li_l:li_u:li_s)$，准确表示出本处理器所负责的迭代集，这就是循环空间重构，它消除了冗余迭代，使结点程序简洁高效。

一般说来，循环空间重构要涉及迭代空间分段和重组等较为复杂的工作[8]，但我们的编译器避免了这些处理，重构过程简单明了：首先根据3.1节关于计算划分策略的讨论，我们知道并行循环语句的局部计算空间是依赖于计算划分标准 CP 来确定的。简单地说，迭代所访问的 CP 数组元素的分布位置决定迭代的执行位置。由3.2.1节知 CP 散布于任一结点的数组片段的全局下标序列一定是规则的，而全局下标一般是循环索引变量的线性函数，所以全局下标序列对应的循环索引序列也是规则的。这就是说，局部循环索引序列一定可以表示为$(li_l:li_u:li_s)$。循环空间重构技术可用图4表示。

图4上面部分是一个一般形式的并行循环语句，其中显示了计算划分标准 CP 在循环体中的位置。下面部分给出这个并行循环语句的结点程序，以说明循环空间重构方法。其中 local_iter 是编译器提供的运行支持函数，参数 DAD_CP 是数组 CP 的 DAD，i_l、i_u 和 i_s 是全局迭代空间的下界、上界和步长，mp 是本处理器的 ID，a 和 b 是 CP 下标关于循环变量的线性函数的系数，运行支持函数 local_iter 根据这些参数首先计算出 CP 在本地数组元素相应全局下标序列，这个全局下标序列一定是规则的，它对应一个规则循环索引序列，local_iter 把这个规则循环索引序列以 li_l、li_u 和 li_s 传回。

...

3.3　通信检测和通信布置

在分布内存多处理器体系结构上，结点程序在各处理器运行时通过通信来访问非本地数据；通信语句由并行编译器生成并布置到结点程序的合适位置。本节首先介绍通信处理的一般策略和算法，再仔细给出我们的通信处理方法，并通过分析和测试比较我们的处理方法的效率和优点。

3.3.1　*通信处理的一般策略及算法*　为方便以下讨论，我们首先明确给出几个概念：

定义1(数组访问)　程序中对某数组的访问表达式称为数组访问，如"A(i,j)"等，可记为 a。

定义2(工作集)　一个并行语句，在计算划分后，某处理器 p 上将被分配一定计算，这些计算涉及的某数组访问 a 的元素集合称为 a 在 p 上的工作集，记为 $Work^a(p)$。

定义3(局部集)　对于某数组 A，其中局部于某处理器 p 的元素集合称为 A 在 p 上的局部集，记为 $Local^A(p)$；一个数组访问 a 局部于某处理器 p 的元素集合也称为 a 在 p 上的局部集，记为 $Local^a(p)$

定义4(通信集)　计算划分后，在某处理器 p 上，某数组访问 a 的工作集 $Work^a(p)$中需要进行通信(不能直接访问)的元素集合称为 a 的通信集。记为 $\sigma^a(p)$。

根据以上定义，我们有：$\sigma^a(p) = Work^a(p) - Local^a(p)$

对任意处理器 q($q \neq p$)，只要 $\sigma^a(p)$中有元素分布在 q 上，在 q 与 p 间就要安排通信动作。这样才能保证结点程序对 $\sigma^a(p)$中元素进行访问。q 与 p 间存在通信的充分必要条件是：$\sigma^a(p) \cap Local^A(q) \neq \Phi$(A 为 a 涉及的数组)

在 $\sigma^a(p)$中，需要在 q 与 p 间进行通信的数据元素组成一个通信子集，记为 $\sigma^a(p,q)$。如果 a 是数据引用，则 $\sigma^a(p,q)$代表接收集，记为 $Recv^a(p,q)$；如果 a 是数据更新，则 $\sigma^a(p,q)$代表发送集，记为 $Send^a(p,q)$；如果 a 既是数据引用又是数据更新(比如某些过程调用实参)，则 $\sigma^a(p,q)$既代表发送集也代表接收集。

对于发送集 $Send^a(p,q)$，将产生发送语句"Send"(比如 MPI 中的 MPI_SEND 函数调用)进行数据发送，同时产生一个接收语句"Recv"(比如 MPI 中的 MPI_RECV 函数调用)进行数据接收；其中"Send"在 p 上执行，"Recv"在 q 上执行。对于接收集 $Send^a(p,q)$也类似产生发送和接受语句，不同的是"Recv"在 p 上执行，"Send"在 q 上执行。

对于一个数组访问，通信处理的关键是计算出通信集，即任意处理器对之间的发送集和接收集。可以通过数组片段的交集来计算通信集。假设某并行循环中存在数组访问"A(i,j)"(记为 a)需要进行通信处理，处理算法如下：

对每个处理器 p 执行：

对每个处理器 q(q≠p)执行：

1. 计算 a 在 p 上的工作集 $Work^a(p)$。

2. 计算数组 A 在 q 上的局部集 $Local^A(q)$

3. 计算通信集 $6^a(p,q)=Work^a(p)\cap Local^A(q)$

4. 根据 a 的访问性质，为 $6^a(p,q)$产生通信语句"Send"或"Recv"，插入结点程序。

3.3.2 *我们的通信处理策略* 上面关于通信处理的一般策略要求编译时计算通信集，对于编译时无法确定通信集的情况不能处理或处理起来非常复杂，比如数组片段的界是变量的情况；这个策略还要求编译器对大量的通信语句"Send"或"Recv"进行组织，以获得高效率，对机器来说这一点是很困难的；另外，直接使用"Send"和"Recv"原语进行通信组织，造成结点程序代码冗长，而且代码生成工作复杂，程序编译变换负担很大。为了克服以上缺点，我们使用运行支持库 Adlib[7]将"Send"和"Recv"等底层通信原语包装起来，做成通信函数，给结点程序提供简明的调用接口；调用通信函数时，只要给出通信数组的分布信息和通信数组片段下标范围等作为参数，复杂的通信动作由通信函数协调完成，这样做的好处在于：

1. 这个策略极大地简化了编译变换工作，使结点程序代码简洁。

2. 可以区分出某些特殊的通信模式，并针对它们进行优化，提高通信效率。

3. 可以在通信函数中人工进行"Send"和"Recv"原语的组织，并使用各种优化策略和技巧，最大限度地提高通信效率。

通信函数在通信形式上是群通信[7]，p_HPF 编译器中的群通信分为无通信、SHIFT 通信和 REMAP 通信[7]。其中 SHIFT 通信是针对整齐边缘通信的通信模式开发的通信函数，根据其特点，我们采用阴影区技术[7]进行优化，使通信效率大大提高。REMAP 通信是除 SHIFT 通信外其它广泛的通信模式，我们在 REMAP 函数中进行了充分的优化，比如利用 MPI 提供的自定义类型进行优化[7]，最大可能地提高效率；测试表明，一个有经验的 MPI 程序员手工编写的通信程序有时赶不上 REMAP 函数的效率，或者效率提高也有限。在我们的策略里，通信处理工作可以分为通信检测和通信布置两个步骤。

我们把所有的通信都提到循环外进行，解决了循环内通信的效率问题；Rice 大学的 dHPF 将循环迭代按访问本地数据还是非本地数据分裂成两部分，单独对访问非本地数据的迭代进行通信（http://www.cs.rice.edu/~dsystem/dhpf/dhpf-overiew-96/index.html）。我们的通信外提策略需要以相关问题处理为补充，有一定复杂性；dHPF 的通信优化策略也很复杂，而且对一些通信模式不支持（http://www.cs.rice.edu/~dsystem/dhpf/dhpf-overiew-96/index.html）。相比之下，pHPF 和 pgHPF 的通信处理要简单得多：由于采用事先将并行循环分裂成多个循环的办法，每个循环内只有一条语句，因此通信可直接外提，不必考虑相关问题。但将并行循环分裂成小循环的策略不够好：首先，每个小循环之间必须进行同步，而同步操作的代价一般都很大；其次，减小了数据重用机会从而降低缓存及寄存器命中率[9]；另外还增加了循环开销，减小了并行粒度。我们的 p_HPF 在处理并行循环时保持循环的完整，不进行分裂。

3.3.3 *通信检测* 通信检测要解决如下问题：结点程序访问哪些非本地数据，这些数据如何分布，采用什么方式的通信（SHIFT 还是 REMAP?）可以将其取到本地或更新。下面给出并行循环语句通信检测问题的一般形式，考虑一维情况，多维情况下只要分别对每维进行类似的考察即可，所以这个形式具有普遍性。

```
      type_x  X(l_x : u_x : s_x)
      type_y  Y(l_y : u_y : s_y)
      !HPF$  PROCESSORS  P(num)
      !HPF$  TEMPLATE  T_x(l_tx : u_tx), T_y(l_ty : u_ty)
      !HPF$  ALIGN  X(I)  WITH  T_x(a_x*I + b_x)
      !HPF$  ALIGN  Y(I)  WITH  T_y(a_y*I + b_y)
      !HPF$  DISTRIBUTE  (dist_type_x)  ONTO  P :: T_x
      !HPF$  DISTRIBUTE  (dist_type_y)  ONTO  P :: T_y
!HPF$  INDEPENDENT
          DO I=Li,Ui,Si
            ...                          CP
            access(X(I))  <-------------
            ...
            access(Y(I))
            ...
          END DO
```

图5 CP 确定的并行循环

这里，type_x 和 type_y 是任意基本数据类型，如 REAL；num 是逻辑处理机个数；dist_type_x 和 dist_type_y 为数据分布类型，可取 BLOCK 或 CYCLIC。假设已通过3.1节介绍的计算划分算法，将 X(I)确定为计算划分标准 CP。通信检测将 Y(I)与 X(I)进行比较，求出 Y(I)的通信方式，采用如下算法：

(1)枚举循环变量 I 的所有取值，根据 X 和 Y 的分布，分别算出 X(I)和 Y(I)所在处理器号；考察对 I 的所有取值，Y(I)是否总是在 X(I)的本地，如是则 Y(I)无通信，否则转(2)。

(2)使用文[3]中的 SHIFT 通信判定定理，判断 Y(I)是否符合 SHIFT 通信条件，如是则 Y(I)为 SHIFT 通信，否则转(3)。

(3)Y(I)采用 REMAP 通信。

```
REAL  A(100), B(100), D(100)
!HPF$  PROCESSORS  P(4)
!HPF$  DISTRIBUTE  (BLOCK)  ONTO  P  ::  A, B, D
!HPF$  INDEPENDENT
         DO  I=2, 99
             A(I)=B(I+1) + F(A(I), c)
             D(I-1)=A(I)  <----------
         END DO                      CP
```

图6 存在数据通信的并行循环

3.3.4 *通信布置* 数组通信方式确定后，编译器要在结点程序中布置适当的通信语句完成通信。对无通信，不需要通信语句；SHIFT 通信和 REMAP 通信分别调用相应的通信函数进行通信。我们把通信函数调用语句简称为通信语句。在结点程序中，通信语句都放在循环（已被重构，见3.2.3节）外。对计算要引用的数据，在循环前设置通信语句进行数据获取；而对计算要更新的数据，在循环后设置通信语句进行数据写回。如图6所示的例子。

图6数组 A、B 和 D 同为 BLOCK 分布，F 是函数调用，c

是标量。假设数组访问 A(I)被选为计算划分标准 CP。通信检测发现 B(I+1)和 D(I-1)需要 SHIFT 通信，其中 B(I+1)是数据获取，D(I-1)是数据更新。SHIFT 通信使用阴影区技术[3]，通信数据的移动如图7所示。

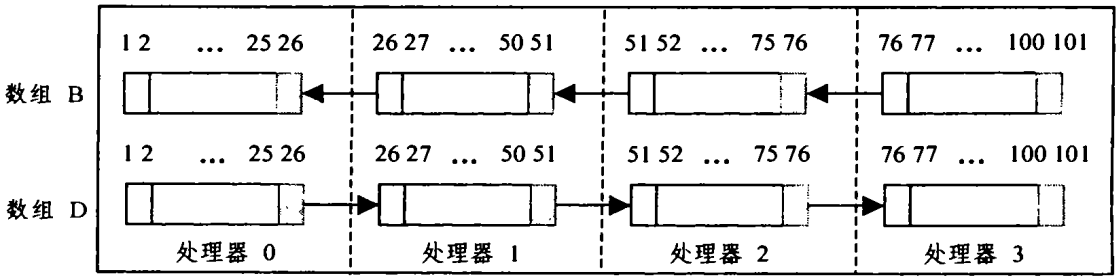

图7 并行循环的数据通信

图7数组 B 和 D 的阴影部分为阴影区[7]，它是数组数据空间的延伸，用来作为 SHIFT 通信的缓冲区；箭头表示通信数据的流动。对数组 B 来说，数据从数据区流向阴影区，为计算准备数据；对数组 D 来说，数据从阴影区流向数据区，完成计算结果的回写。

3.3.5 *通信效率的测试* 为了检验 p_HPF 编译器在通信处理方面的效率，我们作了测试。使用的例程1(jacobi.f)是雅可比迭代程序，其 HPF 版本使用 p_HPF 进行编译，程序涉及 SHIFT 通信；对比程序是文[10]中提供的手写 MPI 版本，该版本代表手写并行程序的权威。使用的例程2(assign.f)是一个数组赋值，其 HPF 版本也使用 p_HPF 进行编译，涉及 REMAP 通信；对比程序使用文[11]中提供的最优化通信策略；该策略主要思想是通过非阻塞通信达到通信与计算时间重叠，从而隐藏通信开销，提高效率；程序也采用 MPI 开发。两个程序的 HPF 版本主要计算部分都是并行循环。测试结果如表1。

表1 (4处理器，100M 以太网连接)

测试程序	数据量	MPI 时间(s)	HPF 时间(s)	MPI/HPF (%)	备注
jacobi.f	2000×2000	10.02	10.39	96.4	HPF 程序采用 SHIFT 通信
	3000×3000	22.75	22.90	99.3	
	4000×4000	39.55	40.17	98.7	
assign.f	500000×16	1.22	1.24	98.4	HPF 程序采用 REMAP 通信
	500000×24	1.83	1.86	98.4	
	500000×32	2.47	2.49	99.2	

表1中 MPI/HPF 一栏是程序的 MPI 版本与 HPF 版本运行时间的比值，也是 HPF 版本与 MPI 版本的效率对比。由这一栏可见，在各个数据规模下，HPF 版本在效率上都十分接近手写 MPI 版本；无论对 SHIFT 通信(jacobi.f)还是 REMAP 通信(assign.f)。这说明并行循环的通信实现效率很高，十分成功。

3.4 相关问题处理

根据并行循环语句的定义，并行循环语句的迭代与迭代之间无数据相关。相关只发生在迭代内部；而计算划分总是把每一个迭代作为一个整体在处理器间分配，所以相关问题本身不会带来通信和同步，可以在本地解决。这是我们计算划分策略的一个显著优点。

并行循环语句相关问题可以一般性地表示为：

```
!HPF$ INDEPENDENT
DO I = i1,iu,is
  ...
  access (A(expr1))
  ...
  access (A(expr2))
  ...
  access (A(expr))
  ...
END DO
```

图8 并行循环语句相关问题的模型

上面并行循环语句中，按执行顺序依次出现若干数组访问；其中 access 表示数据访问，包括引用或更新；A 是数组，expr、expr1和 expr2都是下标表达式，一般为循环变量 I 的线性函数。设结点间通信外提(提出到循环)涉及 A(expr)，且假定在 A(expr)之前，存在另外两个对 A 的访问 A(expr1)和 A(expr2)；我们考虑对 A(expr)的相关处理。由于在 A 为多维数组或在 A(expr)之前出现更多对 A 访问的情况下可以类似处理，上述模型具有普遍性。

在图8的模型中，如果数组访问 A(expr1)、A(expr2)和 A(expr)都无通信，那么也就没有相关问题，因为相关数据的流动由语句顺序执行自动保证，不需要特殊处理。从某种意义上说，是通信(外提)引起了相关问题：上面模型中如果假设 A(expr2)需要通信，而通信被集中安排在循环外，那么循环内对 A(expr2)的访问变成暂时对临时空间的访问，这造成对数组 A 的有关数据不能及时更新，从而 A(expr)无法访问到正确的数据。由此看来，我们的通信处理策略必须以相关问题处理为补充，才能保证结点程序的正确运行。

以上对相关问题根源的分析为我们提供了解决相关问题的思路：只要在循环迭代内插入适当语句，及时完成有关数据的更新就可以了。我们的处理过程分如下几个步骤：

(1)把所有在 A(expr)之前出现的对数组 A 的访问，按执行顺序组成有序元组，记为 S；则图8中有：

S=〈A(expr1),A(expr2)〉

(2)对 S 中的每一元素 S_u，计算当循环变量 I 取哪些值时，S_u 和 A(expr)访问到相同数组元素；我们把满足条件的 I 取值集合记为 $I(S_u)$。

(3)对 S 中的每个元素 S_u，在结点程序中，在访问 A(expr)的语句之前，插入一条语句 $STAT(S_u)$，完成相关数据的流动。如果用 local(X)表示全局访问 X 在结点程序中的局部访问，则对 S 中的元素 S_u，应插入的语句 $STAT(S_u)$为：

```
if (I ∈ I(Su)) then
      local (A(expr)) = local (Su)
end if
```

(4) 假设 S 中有两个元素 S_{u1}和 S_{u2}，其中 S_{u1}在 S_{u2}前，那么 $STAT(S_{u1})$必须插在 $STAT(S_{u2})$之前，因为这才符合顺序执行的语义。

根据上面的策略，图8程序经过相关处理，最终得到如下

形式的结点程序：

```
DO I = li1,li2,li3
    ...
    access (local (A(expr1)))
    ...
    access (local (A(expr2)))
    ...
    if ( I ∈I (A(expr1)) ) then
        local (A(expr))=local (A(expr1))
    end if
    if ( I ∈I (A(expr2)) ) then
        local (A(expr))=local (A(expr2))
    end if
    access (local (A(expr)))
    ...
END DO
```

图9　相关处理后的结点程序

图9中(li_1,li_2,li_3)为局部循环空间，local(X)是X的局部访问形式，可能指向原来数组，也可能指向一个临时空间；I(X)表示使X和A(expr)相关的循环变量I的取值集合；插入的两个if语句完成相关数据的流动，保证计算正确进行。

4　测试和结论

为了检验p_HPF编译器对并行循环语句的总体处理效果，我们使用二维傅立叶变换程序进行了测试。测试所使用的平台是曙光2000，该平台由32个计算结点组成，每个结点配置了PowerPC 640e处理器，256M本地内存和硬盘。并行计算时的通信通过WRC连接的高速Mesh网。操作系统是AIX V4.2，消息通信使用MPICH 1.1。

测试例程1　二维傅立叶变换程序(FFT)，主要计算部分是两个并行循环语句，形如：

```
...
!HPF$ INDEPENDENT
      DO icol = 1, N
          CALL fft_slice (B (:,icol), M, isgn)
      END DO
...
```

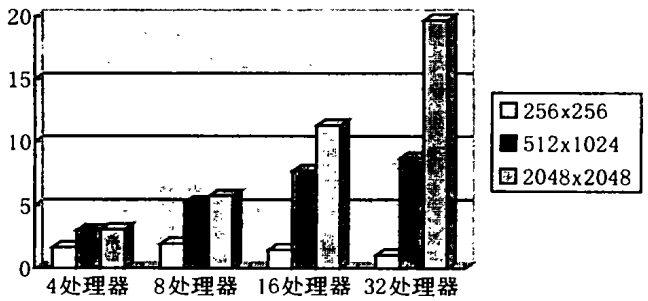

图10　FFT程序加速比图例

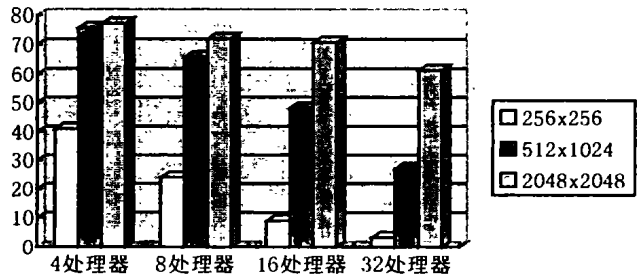

图11　FFT程序效率图例

分析：编译得到的结点程序显示：编译器对该程序中并行循环语句的计算划分达到无通信需求且负载均衡的理想结果。虽然整个程序中有两次remap通信(两个TRANSPOSE)，时间开销较大，但主要计算部分并行循环语句处理得较好，程序加速比较高。

处理器个数一定的情况下，小数据量加速比较小，效率不明显，这是因为计算量较小时，通信和运行支持函数开销相对较大。当数据达到一定规模时，加速比较高，效率达70%左右，并行编译获得了良好的并行效果。表明本文的编译方案具有很好的效率。

对相当规模的数据量，加速比随处理器个数的增加而显著提高，比如对2048×2048的数据，处理器数为4，8，16，32时，加速比分别达到3.09，5.76，11.30，19.65。表明我们的编译方案具有良好可扩展性。

为了把我们的编译处理与其它算法进行对比，我们选择SPEC92的nasa7标准测试程序进行测试，并与文[5]和文[6]中的测试结果比较。

测试例程2　cholesky

```
DO 1 J = 0,N                                              (L1)
  I0=MAX(-M,-J)
  DO 2 I = I0,-1                                          (L2)
    DO 3 JJ = I0 - I, -1                                  (L3)
    DO 3 L = 0,NMAT                                       (L4)
3     A(L,I,J) = A(L,I,J)- A(L,JJ,I+J)* A(L,I+JJ,J)
   DO 2 L = 0,NMAT                                        (L5)
2     A(L,I,J) = A(L,I,J)* A(L,0,I+J)
  DO 4 L = 0,NMAT                                         (L6)
4 EPSS(L) = EPS* A(L,0,I+J)
  DO 5 JJ=I0,-1                                           (L7)
   DO 5 L = 0,NMAT                                        (L8)
5  A(L,0,J)=A(L,0,J)- A(L,JJ,J)**2
   DO 1 L = 0,NMAT                                        (L9)
1  A(L,0,J) = 1. / SQRT ( ABS(EPSS(L) + A(L,0,J)) )
   DO 6 I = 0,NRHS                                        (L10)
    DO 7 K = 0,N                                          (L11)
     DO 8 L = 0,NMAT                                      (L12)
8     B(I,L,K) = B(I,L,K)* A(L,0,K)
   DO 7 JJ=1,MIN(M,N-K)                                   (L13)
     DO 7 L = 0,NMAT                                      (L14)
7      B3(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ)* B(I,
       L,K)
    DO 6 K = N,0,-1                                       (L15)
     DO 9 L=0,NMAT                                        (L16)
9     B(I,L,K) = B(I,L,K)* A(L,0,K)
     DO 6 JJ = 1,MIN(M,K)                                 (L17)
      DO 6 L = 0,NMAT                                     (L18)
6       B(I,L,K-JJ) = B(I,L,K-JJ)-A(L,-JJ,K) * B(I,L,K)
```

以上代码由18个不规则循环嵌套组成，其中一些计算的嵌套深度达到4。在文[5]中被作为并行化的难点给出。

实际上，该代码中所有以L为循环变量的循环都是并行循环。我们的编译方案能够抓住这些循环的并行特征，对所有的L循环进行划分而其它循环不动，从而达到对整个计算任务的划分。因为对每个L循环的计算划分都能达到负载均衡且无通信，所以整个计算过程也达到了负载均衡且无通信的最优并行效果。测试结果表明我们获得了很高的加速比，8处理器达到7.0。

文[5]中提出的仿射计算划分(Affine Partition)算法，通过幺模变换、循环融合、以及循环交换和融合交替使用等技术，对例2达到与我们相类似的划分方案，加速比略低于我们，8处理器达到6.2。相比之下我们的编译处理更简单有效。改进的仿射计算划分算法对该例处理结果出现在文[6]，加速比没有太大变化，8处理器达到6.3。但应当承认，仿射计算划分算法可以对非并行循环进行并行化，这一点是我们编译处理框架所不能的。

其它算法如幺模变换法(unimodular transform)[3,4]或数据计算分解加障碍同步消除法[4]等对上例的处理都不理想，加速比很低。比如幺模变换法将对L10循环进行并行化，因为它是最外层并行循环，但实际上这个方案带来很大通信开销；

对这个例子，幺模变换法不能发现最优划分方案。

为了进行比较，我们给出以下四种算法及对例2的测试结果：

1. 幺模变换法
2. Anderson 和 Lam 的数据计算分解加障碍同步消除法
3. 仿射计算划分(Affine Partition)算法
4. 我们的编译处理框架。

前三种算法的测试结果由 Stanford 大学计算机系统实验室在文[5]给出，测试平台是 Digital Turbolaser，使用8个300-MHz 的21164处理器。我们的测试平台是100M 以太网连接的8个 PC 机，结点机使用 P III 550处理器。我们的平台更简易，所以我们的编译处理获得了更高的性能价格比。测试结果如图12所示。

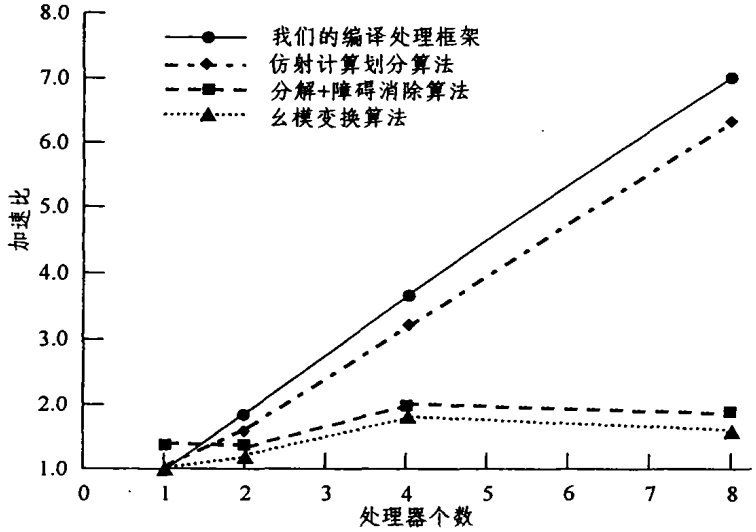

图12 SPEC92的 nasa7标准测试程序性能

测试表明，我们开发的 p_HPF 编译器对 HPFBench[12]和 NAS Parkbench[13]等重要 benchmark 程序中所出现的并行循环语句全部支持，达到理想的并行效果，效率很高。另外，从测试例子可以看出，我们实现的并行循环语句具有相当强的处理能力，完全可以处理实际应用中复杂的并行计算需求。p_HPF 对石油地震资料处理典型应用程序的支持证实了这一点。我们把石油地震资料处理典型应用并行化，使用 HPF 语言编程，并用 p_HPF 编译运行，结果获得了良好的加速比和效率，并行化工作很成功；我们也因此获得胜利油田的支持，进行石油软件并行应用研究；在这个工作中，p_HPF 所实现的并行循环语句发挥了较大作用，表现了简洁有力的计算表达能力。下一步将在此基础上，实现并行循环语句对非规则数组的支持，满足实际应用的需求，这项工作现在正在进行。

参 考 文 献

1 Calzarossa M, Massari L, Tessera D. Performance issues of a HPF-like compiler. In Future Generation Computer Systems, 2001,18:147～156

2 黄其军，杨建武，等. 基于规范划分集的并行循环计算划分算法. 2001

3 Banerjee U. Unimodular transformations if double loops. In: Proc. of the Third Workshop on Languages and Compilers for Parallel Computing, Aug. 1990. 192～219

4 Banerjee U. Loop Transformations for Restructuring Compilers. Kluwer Academic, 1993

5 Lim A W. Cheong G I, Lam M S. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In: Proc. of the 13th ACM SIGARCH Intl. Conf. on Supercomputing, June, 1999

6 Lim A W, Liao S-W, Lam M S. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, June, 2001

7 汪剑平，都志辉，等. HPF 编译器的通讯策略与相关算法. 计算机学报，1999. 5

8 Code Generation and Optimization for High Performance Fortran. Louisiana State University, Ashwath Thirumalai, Aug. 1995

9 Wolfe M. High Performance Compilers For Parallel Computing. Addison-Wesley Publishing, 1996

10 都志辉. 高性能计算并行编程技术——MPI 并行程序设计. 清华大学出版社，2001年8月

11 Benkner S, Zima H. Compiling High Performance Fortran for distributed-memory architectures. Parallel Computing, 1999, 25: 1785～1825

12 HPFBench: A Hign Performance Fortran Benchmark Suite. Harvard University, Rice University, Nov. 1999. http://dacnet.rice.edu/Depts/CRPC/HPFF/benchmarks/index.cfm

13 Bailey D, et al. The NAS parallel benchmarks 2.0: [Technical Report NAS-95-020]. NASA Ames Research Center, Dec. 1995