计算机科学2000Vol. 27№. 1

Java自动并行编译系统JAPS中表达式分离技术的研究*)

A New Technology on Expression Trasformation in JAPS

王 征 方 斐 陈道蓄 谢 立

(南京大学计算机软件新技术国家重点实验室 计算机科学与技术系 南京210093)

Abstract The technology about expression transformation is a very important work during the pre-process phase of automatic parallelizing compiling. The source program can fit the system more after the processing. Expression transformation' s emphases vary in different systems. This paper put forward a new technology on expression transformation aim at NOW-based JAPS(Java Automatic Parallelizing Compiling System), during which method-call can be separated from expression based on the principle of correctness, proper granularity and simpleness.

Keywords Automatic parallelizing compiling, Expression transformation, Granule, Readset, Writeset

一、引言

1.1 自动并行编译的提出

并行程序设计基本上有两种途径:显式途径和隐式途径[1],研究者设计了很多的并行程序设计语言,这就是所谓的"显式途径"。但是,并没有一种并行程序设计语言成为流行的语言,主要的问题在于并行程序的编制困难,对程序员的要求高。

相比较而言,隐式途径,即自动并行编译技术就有许多的优势。用户使用串行语言编制程序,由自动编译程序完成并行性的识别,程序的转换,并行代码的生成以及在并行环境下的调度执行。自动并行编译的研究还有另外一个重要的意义,即软件的继承性问题。在高性能计算领域人们已经编写了大量的串行程序,这些程序在向新的软硬件计算平台上移植时,会耗费大量的人力和物力,自动并行编译将大大减轻这方面的劳动量。

自动并行编译及相关并行化工具在并行分布系统(如MPP,NOW)和高性能计算领域的应用软件之间架起了一座桥梁,它一方面使用户将更多的精力放在程序自身的设计上,另一方面又能发掘应用程序潜在的并行性,充分利用硬件系统性能。

1.2 Java自动并行编译的提出

Java语言是一种新兴的面向对象的程序设计语言,其最大的特点是平台无关性。此外,它还有多线程、无用内存的搜集等特点[2]。随着Internet的迅速普及,Java也得到了广泛的应用,在高性能计算领域也开始使用Java作为编程语言,因此,以Java语言作为目标语言进行面向对象程序设计语言的自动并行编译研究是有必要和有意义的。

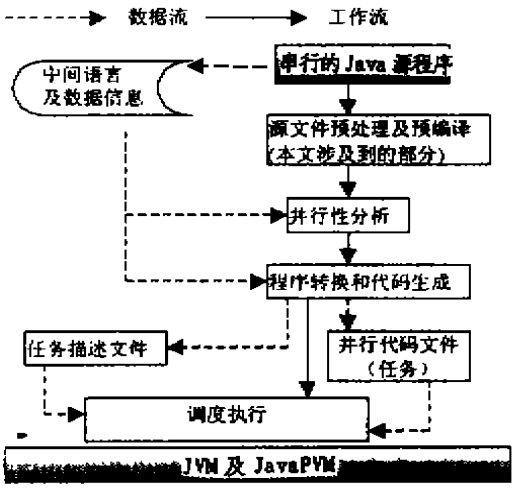

图1 JAPS模型

1.3 Java自动并行编译工具集JAPS的设计

在NUAPC 1,NUAPC Ⅱ中[3,4],我们以C++为

*)得到国家攀登计划B和863课题支持。

目标语言进行了 OOPL 的自动并行编译研究。在此研究基础上，我们设计了一个基于 NOW 的 Java 自动并行编译系统 JAPS(Java Automatic Parallelizing System)[5]。JAPS 提供了一组编译工具，将串行的 Java 源程序进行预处理，通过分析转换，生成并行 Java 代码，然后在 JavaPVM 和 NOW 环境下运行。

图1给出了 JAPS 的总体模型。

在该模型中，既可以进行传统的数据并行型的分析，也可以进行任务并行性的分析。

Java 源程序并行化工作的第一步是对其进行预编译，程序流图的构造与分析以及源程序的处理(包括表达式处理)是预编译阶段的主要组成部分。

本文所涉及的工作是表达式处理，是 JAPS 中源程序的预编译和预处理模块的一个子功能，该模块的其余部分和 JAPS 的其余模块限于篇幅，就不做过多的描述。

二、分离技术的研究

在并行性分析中，如果任务基于语句，而不是基于表达式则会使任务的划分和封装更加简单，并且可以方便地找到一个任务对应的用户输入的源程序部分。但是，在任务划分中，常常遇到一个简单语句的粒度较大，该简单语句自己就会形成一个任务，如果该任务的粒度大于希望的粒度，就不能通过重新组合和划分任务内的语句来发掘更深的并行性。经过分析，这种简单语句往往包含有方法调用，方法调用的粒度是导致该简单语句的粒度大的主要原因。在一个简单语句中，可能会有多个方法调用，如果把一些方法调用从原语句中分离出来，单独组成一条简单语句，则可以有效地降低大粒度的简单语句的粒度。这就是表达式分离。

因此，使用表达式分离可以充分发掘任务之间的并行性，但是不能无限制地进行表达式分离，否则会带来临时变量使用的膨胀。因此，应当探讨表达式分离的原则。

表达式分离的原则主要有正确性原则，粒度适中原则和简单性原则。在介绍这些原则前，首先介绍一下数据的读写集，粒度。

2.1 数据读写集

数据流的分析基础是确定节点的数据读写集。一般来说，程序流图中的一个节点 N 访问的数据可分为如下两种基本集合：

(1)写集　记为 MOD(N)：对于一个节点 N 及一个变量 v：v∈MOD(N)表示执行 N 后改变了 v 的值。

(2)读集　记为 USE(N)：对于一个节点 N 及一个变量 v：v∈USE(N)表示执行 N 后使用了 v 的值。根据读写集，就可以得到语句间的依赖关系[6]。

2.2 粒度

把一个 Java 源程序的语言结构按级别依次分为：变量(声明)、表达式、语句、语句组、方法、类和接口、程序包。程序并行性就是从这些不同粒度的层次概念出发测试相关的依赖关系得到的。

粒度是程序计算时间的量度。大粒度并行程序用大部分时间进行计算而不经常通信，细粒度并行程序则经常通信。在松散耦合系统或现有网络速度相对较低的 NOW 环境中，中、细粒度的并行会造成非常大的通信开销，故并不合适，而大粒度并行应用对上述系统是非常合适的。

对于 Java 这样的面向对象语言，可以考虑进程/线程、对象、方法/任务(procedure/task)、语句几种并行粒度的选择。

在 JAPS 中，以其中的方法/任务并行性作为选取并行粒度的标准。这种中、大粒度的任务在现有的实现环境，包括 NOW，JavaPVM 等上体现了很好的性能。

最终的并行粒度是与系统状况紧密结合在一起的。在确定任务的并行粒度时，会对任务并行粒度进行一些调整，把一些并行粒度较小的任务合并成一个大的任务，或者把一个并行性粒度较大的任务划分开，形成几个粒度相对较小的任务。

2.3 表达式分离的正确性原则

表达式分离的实现基础在于在表达式中，方法的计算地位与一个变量的计算地位基本相同，因此，如果把一个方法用一个与该方法的返回值具有相同的值的变量代替，表达式的值一般与原表达式相同。例如：a+f(b,c)+d 与 a+e+d(其中 e=f(b,c))的值一般是相等的，因此可以把原表达式转换为：e=f(b,c);…a+e+d…；但是有时两者是不相同的，经过分析，这种现象一定发生在以下几种情况：

(1)方法修改了非方法的局部变量的变量(如：域变量，形式参数)。

例1

```
class class_A{
    int a=1;
    int function_a(int b){return(a++)+b;}
    int function_b(){
        return a+function_a(1)+a;
    }
}
```

例1中，function_b 中的表达式 a+function_a(23)+a 的值为：1+(1+1)+2=5，但是，如果把该表达式变为：int b=function_a(1);return a+b+a;则 a+b+a 的值为：2+2+2=6。

(2)调用方法时方法的参数被修改。这种现象有三种情况，分别举例如下：

例2

```
class class_A{
      int function_a(int b){return b+1;}
      int function_b() {
            int a=1;return(a++)+function_a(a);
      }
;
```

例2中，function_b 中的表达式(a++)+function_a(a)的值为：1+(2+1)=4，但是，如果把该表达式变为：

```
int b=function_a(a);
return(a++)+b;
```

则(a++)+b 的值为1+2=3。

这种情况下，导致两个表达式的值不一样的原因是在原表达式中，被调用的方法的参数涉及的变量在调用该方法前被修改，调用该方法时作为参数传给方法的值为修改后的变量的值，而在变形后的表达式中，被调用的方法的参数涉及的变量在调用该方法后被修改，调用该方法时作为参数传给方法的值为修改前的变量的值，因此，导致变形前后的值出现不一致性。

例3

```
class class_A{
      int a=1;
      int function_a(int b){return b+1;}
      int function_b() {
            return a +function_a(a++)+a;
      }
;
```

这时表达式 a+function_a(a++)+a 的值为：1+(1+1)+2=5，如果把表达式变为：

```
int b=function_a(a++);return a+b+a;
```

则 a+b+c 的值为2+2+2=6。这是因为表达式的非函数调用部分使用了与方法调用参数部分相同的变量，而该变量在调用方法后被修改。

例4

```
int function_a(int & a){return a++;}
int function_b(){
      int b=();
      return b+function_a(b)+b;
}
```

这个例子并不适合 Java 程序，因为 Java 的方法调用不容许使用形参，这里列出这个例子是为了说明表达式方法调用外提的过程中要注意的问题。该例子为 C++ 程序，因为使用了形式参数，产生了函数的副作用，这时表达式 b+function_a(b)+b 与变形后的值不一样。

以上所举的情况都有一个共同点，可用以下的公式表示：

$$(MOD_A)F)\cup MOD_B(F))\cap MOD_C(F))\cup((USE_A(F)\cup USE_B(F))\cap MOD_C(F))\cup((MOD_A(F)\cup MOD_B(F))\cap USE_C(F))\neq\varnothing \quad (1)$$

其中，对于表达式中的一个被调用方法 F，设：

$MOD_A(F)$：为方法 F 的实现部分的变量写集。

$MOD_B(F)$：为表达式中调用方法 F 时的参数的变量写集。

$USE_A(F)$：为方法 F 的实现部分的变量读集。

$USE_B(F)$：为调用方法时的参数的变量读集。

$MOD_C(F)$：为原表达式除去该方法调用部分的其余部分的变量写集，该写集包含有其余被调用方法的实现部分的变量写集。

$USE_C(F)$：为原表达式除去该方法调用部分的其余部分的变量读集，该读集包含有其余方法调用的参数的变量读集。

如果一个方法调用满足以上的条件，则不应把该方法调用外提，以免造成表达式计算错误。如果一个表达式的一个方法调用不满足以上的条件，则表示该方法调用可以被外提。

例5

```
class A{
      int value;
      void function_a(){
            A a,b;
            a=new A();
            b=a;
      }
}
```

在分析读写集时，应注意到一种变量的复制情况，例如例5，当执行到 b=a 后，a. value 与 b. value 代表的变量应是同一个变量，因此，这时为了使式1中的读写集精确，应使用数据流的分析方法[7]分析出表达式的较精确的读写集。对于无法分析出读写集的表达式，不能实行表达式分离技术。

2.4 表达式分离的一些特殊情况——表达式分离的准确性的再探讨

还有一些情况下，虽然方法遵循以上的原则可以提出来，但是会给程序的整体安排带来一些影响。另外，在 Java 语言中如果提出循环语句的判断条件中的方法调用，则可能无法重写循环语句。在这种情况下，不应当随便进行表达式的分离。这种特殊的表达式主要是以下的几种表达式：

(1)do 语句，while 语句的判断条件语句；

(2)for 语句的判断条件语句，for 语句的递增语句。

这些情况下，如果把方法调用提出，则需要把原表达式分为两个表达式，而在循环语句中，条件判断语句不能为两个表达式，因此，必须寻找办法把循环语句重写，在支持跳转语句或逗号表达式的语言中，可以用跳转语句或逗号表达式改写，但是在 Java 语句中，却没有跳转语句和逗号表达式，因此无法简单地重写循环语句。

例如：例6中，如果使用 C++ 语言，可以有以下的两种改写形式：

改写形式1：使用逗号表达式，见例7。

改写形式2：使用跳转语句，见例8。

例6

```
while (function_a(23)+b){
    if(c>20){
        c=c+2;
        continue;
    }
    c=c+1;
}
```

例7

```
while (temp_var=function_a(23),temp_var+b){
    if (c>20){
        c=c+2;
        continue;
    }
    c=c+1;
}
```

例8

```
while(true){
label_con:
    temp_var=function_a(23);
    if(!(temp_var+b)){
        break;
    }
    if(c>20){
        c=c+2;goto label_con;
    }
    c=c+1;
}
```

研究改写形式，会发现只有当循环体中出现循环内部的 continue 语句后，改写才会用到 goto 语句。此外，如果出现 break 语句，改写并不会用到 goto 语句，因此，如果一个 Java 程序的循环体中不出现在该循环内部的 continue 语句，则可以使用改写形式2以 break 来改写原循环语句。

2.5 表达式分离的粒度原则和简单性原则

分离方法时应考虑方法的粒度，如果一个方法的粒度很小，则没有分离的必要，这是分离的粒度原则。在并行性分析中也有粒度分析，在分离方法阶段只要能够大致区分方法的粒度大小就可以了，这是简单性原则。一些粒度分析不足的情况可以在并行性分析阶段补足。对于任务划分阶段，任务合并总是要比任务分割要简单，因此在一些不能得到粒度的情况下，则总是进行表达式分离。

方法的粒度确定有时并非容易，如一些递归方法、一些带有判断语句或循环语句的方法往往不能确定递归层数或循环次数，对于这种情况，不易确定方法的粒度大于预想值。但是在一些简单情况下，可以确定循环次数或递归层数，下面给出了确定语句（包括方法）粒度的原则和方法：

(1)*表达式的粒度*：表达式的粒度为该表达式中所有的运算的运算时间的总和，包括调用的所有方法的运算时间，算术运算的执行时间，以及一些附加时间，如类型转换花费的时间等，如果其中有一个运算时间为无穷大，则该表达式的粒度也为无穷大。

(2)*判断语句的粒度*：分 if 语句的粒度和 switch 语句的粒度。

(a)if 语句的粒度：对于 if(a)b;else c，这里 b，c 可以为复合语句，b，c 的粒度可以通过语句的粒度确定原则确定。设 a 的值为真的概率为 p_a，a，b，c 的粒度分别为 L_a，L_b，L_c，该语句的粒度可以粗略估计为：$L_b * p_a + L_c * (1-p_a) + L_a$。如果 a 的值可以确定，则 $p_a=1$ 或 $p_a=0$；如果无法得到 p_a，可以认为 $p_a=0.5$，表示执行 b，c 的几率各为0.5，这时该语句的粒度为 $(L_b+L_c)/2+L_a$。

对于 if(a)b;语句，可以认为 $L_c=0$ 来确定该 if 语句的粒度。

(b)switch 语句的粒度：对于 switch 语句，假设共有 n 个 case 语句，每个 case 语句的粒度为 $L_i(1\leqslant i\leqslant n)$，default 语句的粒度为 L_{n+1}，判断语句的粒度为 L_s，判断语句的值为第 i 个 case 语句的判断条件的几率为 p_i，则该 switch 语句的粒度可用以下的公式粗略表示：

$$L_{switch}=L_s+g_1*L_1+\cdots+g_i*L_i+\cdots+g_{n+1}*L_{n+1} \quad (2)$$

其中 g(i)使用以下的方法确定：

$$g_i=p_m+p_{m+1}+\cdots+p_{j-1}+p_i \quad (3)$$

其中：$p_j(m\leqslant j<i)$ 满足：第 j 条 case 语句中 break 或 return 语句不是该语句的最后一条执行语句，而第 m-1条 case 语句则以 break 语句或 return 语句作为该语句的最后一条语句。在这里，为了简单化，忽视了 break、return 出现在 case 语句中间部分的情况。

(3)*循环语句的粒度*：由于 do、while、for 语句中可以出现 break、continue 语句，甚至在 Java 语言中，break、continue 语句可以跳转到一个标号处，这对于循环体的粒度带来了很大的不确定性。但是，在常见的科学计算中，循环语句一般非常规整，循环的次数可以分析出来，因此可以只确定规整的循环语句的粒度，对于无法确定粒度或比较难确定粒度的循环语句，可以让该语句的粒度为无限大。以下给出了各种循环语句的一些简单情况下的粒度确定方法：

(a)do 语句、while 语句的粒度：可以容易确定的 do，while 语句粒度的形式定义如下：

①循环语句的条件判断表达式满足以下的条件：ⓐ判断表达式中涉及的变量只有一个变量出现在循环体的写集中。ⓑ判断表达式的形式为：A opt B，其中：A 或者 B 为满足条件ⓐ的变量，opt 为一个双目比较运算符，如果 A 为满足条件ⓐ的变量，则 B 可以在编译阶段得到结果，如果 B 为满足条件ⓐ的变量，则 A 可以在编译阶段得到结果。

②在循环体中，对于条件判断表达式中满足上面条件的变量的修改应是可以明确地推算出的，而且，在

循环体中，应该没有 break 和 continue 语句。

(b)for 语句的粒度确定与 do，while 语句基本相同，但是考虑到递增表达式的计算粒度，循环体的粒度应为 for 语句的循环体的粒度加上递增表达式的粒度。此外，循环体应认为是 for 语句的循环体与递增表达式的并集。

满足以上条件的循环语句可以确定循环变量的起始值、终止值以及步进值，因此，这种循环语句可以确定其粒度。

(4)复合语句的粒度：虽然一个复合语句中可能有 break、continue、return 等跳转语句，但是在前段分析阶段的粒度确定中，一个复合语句的粒度可以认为大致是该复合语句的所有的语句的粒度之和。

(5)一个表达式中出现未知粒度的方法调用的时候该表达式的粒度计算：在一个方法实现中一般都有方法调用的情况，而调用一个方法时，往往被调用的方法的定义并不在调用方法的前面声明，即调用该方法时并不知道该方法的粒度，这时原调用方法的粒度也无法立即得出。这里给出了一个简单的方法粒度的确定算法：

如果一个程序段中涉及了 n 个方法(包括调用方法、被调用方法)，它们的粒度记为：$t_1, t_2, \cdots, t_n$，设在这 n 个方法中，有 m 个方法在该程序段中被定义，不妨设为：$t_1, t_2, \cdots, t_m (m \leqslant n)$。则 $t_{m+1}, \cdots, t_n$ 中的一部分可用手工确定其粒度，例如库方法可以参考实现大致确定其粒度，其余的为了节省工作量或没有源程序可以设为无穷大。

在源程序扫描过程中构造一个方程组，该方程组的每一个方程为：

$$t_i = f_i(t_1, t_2, \cdots, t_n)(1 \leqslant i \leqslant m, t_j \geqslant 0, 1 \leqslant j \leqslant n) \quad (4)$$

该方程组为 m 元一次方程组，有 m 个未知量 $(t_1, \cdots, t_m)$。

当分析结束后，开始处理该方程组，在处理该方程组时可以使用如下的方法：

当方程 $t_i = f_i(t_1, t_2, \cdots, t_n)(1 \leqslant i \leqslant m)$ 的右边部分合并同类项后，一定有以下的形式：

$$t_i = m_1 t_1 + \cdots + m_i t_i + m_{i+1} t_{i+1} + \cdots + m_n t_n$$
$$(m_j \geqslant 0 (1 \leqslant j \leqslant n)) \quad (5)$$

①如果 $m_i \geqslant 1$，则表示在用户的程序中可能存在有无限递归。这时，t_i 应该设为无穷大，而且应该通知用户在该程序段中可能有无限递归。同时，应当设置直接或间接调用第 i 个方法的方法的粒度为无穷大。

②如果 $t_1, \cdots, t_n$ 都为常数，则 t_i 可以被计算出来，并且 t_i 一定大于或等于零。

③如果 $t_1, t_2, \cdots, t_{i-1}, t_{i+1}, \cdots, t_n$ 中只要有一个的值为无穷大，则 t_i 的值为无穷大。

由于该方程组是 m 元一次方程组，则根据以上的原则，可以得到 $t_i (1 \leqslant i \leqslant m)$ 的值，并且 $t_i \geqslant 0 (1 \leqslant i \leqslant m)$，$t_i$ 可能为无穷大。

通过以上的原则就可以确定所有方法的粒度。对于方法粒度大于希望值(包括粒度为无穷大)的方法调用，要采用表达式分离，对于方法粒度小于希望值的方法调用，则不应把方法调用分离出原表达式。

三、系统实现

1. 实现环境

本系统的运行硬件环境是由多台个人计算机、多台 RS6000工作站及多台 Sun Sparc 工作站通过 EtherNet、ATM 网构成的分布式并行计算环境。采用的操作系统包括 Microsoft Windows 95，Microsoft Windows NT 4.0，IBM OS/2 3.0，IBM AIX 4.2，Sun OS5.x。

本系统的源程序预处理部分采用 LEX，YACC 为辅助工具开发，使用标准 C 语言以便于平台间的移植。调度器采用 Java 语言实现，一方面从系统的执行角度看，调度程序与并行 Java 程序在同一层次；另一方面，利用 Java 的平台无关性，使整个系统便于移植。

2. 表达式变形的程序实现

在 Java 的语法中，允许存在先定义后使用的情况，如类的方法或域可以在未定义之前在域的初始化中使用，或者在方法定义中使用。因此为了准确地搜集信息，在表达式变形处理中采用了多趟扫描的方法，下面简要介绍一下各趟扫描的作用：

(1)第一趟扫描：搜集一些定义信息和生成源文件的结构信息，包括类的定义信息，方法的定义信息，变量的定义信息，类的派生关系信息，continue、break、return 语句的使用，以及有关的语法树。

(2)第二趟扫描：主要是在第一趟扫描的基础上搜集使用信息。包括方法的使用信息，变量的使用信息，节点的粒度信息、以及变量的可能取值。当该次扫描结束后，根据语法树和第一、第二趟扫描得到的信息，根据方法分离的实现基础和方法分离的特殊情况，找到可以分离的方法调用，然后根据节点的粒度信息，算出方法调用的粒度从而决定哪些方法调用最终需要与原表达式分离。

(3)第三趟扫描：主要是进行表达式中方法调用的分离工作。根据第二趟扫描的分析结果，进行最终的方法分离工作，生成一个处理后的 Java 源文件，在该文件中，粒度较大的可以分离的方法调用被从原表达式中分离出来。

这三趟扫描中只有第三趟扫描是与表达式变形密切相关的，第一，第二趟扫描是Java源文件分析的基

计算机科学2000Vol. 27№. 1

可移动代码的一种优化技术

An Optimization Technique for Mobile Code

王明文　孙永强

（上海交通大学计算机科学与工程系　上海200030）

Abstract　Partial evaluation provides a unifying paradigm for a broad spectrum of work in program optimization, compiling, interpretation and the generating of automatic program generators. To improve the performance of mobile codes system, we use the partial evaluation technique to specialize the mobile code in the server, which can speed up the running of the mobile code in clients and decrease the communication quantity in the network.

Keywords　Mobile code, Partial evaluation, Program optimization

1. 引言

关于可移动的程序代码的研究起源于异构网络环境的需求。象 Scheme[2]，Tcl[3]和 Telescript[4]这样的解释型语言都在一定程度上支持程序代码的可移动性。和移动代理的迁移所不同的是，可移动代码只是程序代码的迁移而不是运行程序（包括程序代码、数据以及运行状态）的移动。最为我们所熟悉和使用的可移动代码当然是由 Sun 微系统公司开发的 Java 语言[5]了。Java applets 和 Java servlets 是目前使用最为广泛的可移动代码技术。

Java applets 是用于 WWW 网页的 Java 程序。当用户使用内置 Java 解释器的浏览器浏览 Web 站点上的网页时，网页服务器上的程序被自动下载到用户的本地机器上并被装载执行。由于程序的执行是在浏览者的本地计算机上进行，也就没有了网络延迟，因此程序可以给用户提供复杂的图形化界面并且能对用户的动作作出快速反应。不受信任的 applets 程序只能在安全的 Java 解释器中解释执行，因而它不能存取或破坏敏感信息。Metis 的瘦客户（Thin-client）框架可以被看成是 Java applets 的一般化。在 Metis 中任意客户可以下载一个用 Java 写的应用的前端程序，它负责寻找一个或多个实现整个应用的网络服务并与之进行交互以完成任务。其它支持 applets 的系统还有：CNRI 提供的支持用 Python 写的 applets 程序的 Web 浏览器 Grail；Sun 微系统公司提供的支持用 Tcl/Tk 写的 ap-

础，扫描方法可以提供给 JAPS 的其余模块使用。

总结　本文所涉及的项目 JAPS 属于国家攀登计划 B 的课题“大规模并行编译和操作系统的某些关键技术”的一项研究内容，同时受到了“863”课题“基于 Java 的使用化分布并行工具包”的支持，已经通过了国家863项目鉴定。

在系统设计和实现中，还存在着一些问题，如与并行性分析的结合不是太紧密，导致一些分析结果不够准确；数据流的分析还不太完善；简单性原则不易确定；判断语句的分支概率的确定方法、粒度分析的方法等还有待探讨；并且没有实现表达式的运行时刻的动态调整。为了解决这些问题，还要进行进一步的分析和探讨。

致谢　感谢杜建成博士，对本文工作提出了许多有益的启发。

参考文献

1　张德富. 并行处理技术. 南京大学出版社，1992. 13～15
2　Gosling J, McGilton H. The Java Language Environment: A White Paper, 1995
3　Zhu Genjiang, et al. PRG: Procedure Reference Analysis in Extracting Non-data Parallelism. In: 1996 IEEE Second Interl. Conf. on algorithms & Architectures for Parallel Processing. Singapore, 1996. 76～83
4　Zhu Genjiang, et al. A path-based method of parallelizing C++ programs. ACM SIGPLAN NOTICES, 1994, 29(2). 54～64
5　杜建成. 基于 NOW 的面向对象语言自动并行化中几个问题的研究：[博士论文]. 南京大学，1999
6　[美]Banerjee U. 超级计算中的依赖关系分析. 沈志宇，赵克佳译. 湖南科学技术出版社，1991. 26～42
7　Aho A V, et al. Compilers, Principles, Techniques, and Tools. Addison-Wesley Publishing Company, 1996. 660～694