维普资讯 http://www.cqvip.com

自动并行编译 层次分析 并行处理 (10) 编译程序

计算机科学1998 Vol.25 №.6

自动并行编译中的层次分析技术

41~44

Hierarchical Analysis Technique in Parallelizing Compiler

徐 融 朱根江 谢 立

(南京大学计算机系 南京210093)

TP314

摘 要 Hierachical Task Graph(HTG) is an efficient intermediate representation in parallelizing compiler. It decomposes program into hierachical tasks based on strongly connected regions, which can be used to exploit coarse-grained parallelism. Compared with other hierachical analysis techniques such as interval analysis etc., it is of large node number. This paper reduced the node number in HTG by combining the interval analysis method with HTG technique.

关键词 Hierarchical Task Graph, Hierarchical analysis, Interval analysis, Parallelizing compiler

1 引言

当今,绝大部分的超级计算机都采用了某种形式的并行处理技术。在并行机上进行程序设计,有两种选择:一是设计全新的并行语言,如Linda, Occam等;或者在传统的串行语言中加入并行的成分,如PC^{++}[7~9],CC^{++}[6]等。二是让用户仍采用传统的串行语言来编制程序,由自动并行编译来实现程序的并行化。前一种方法存在软件的继承性问题,现有的大量由串行语言编制的软件要在并行机上取得好的运行效果,要花费大量的人力、物力来重新设计、构造现有软件。另外,并行程序设计远较串行程序设计复杂,更难保证正确性,大多数的程序员都宁愿采用串行程序设计。因此,自动并行编译在并行研究领域有着非常重要的意义。

自动并行编译可以分成两个阶段:1. 依赖关系分析2. 程序转换与并行代码生成。这两个阶段的工作相对独立。第一个阶段所做的工作主要是对源程序进行控制依赖与数据依赖分析,然后将两种分析结果综合并优化,得到程序的依赖关系。这部分的工作与机器无关,是并行编译着重研究的领域,自七十年代以来进行了大量的研究。第二阶段利用依赖关系把源程序分为一个个相对独立的模块,并提供一执行模型协调这些模块的执行。以前的依赖关系分析集中于数据并行性的挖掘,是针对循环和语句一级的并行化,这一级的依赖分析技术已经基本成熟。近年来,在并行体系结构以及对过程分析技术的研究所取得的进步,为自动并行编译开辟了新的方向,即对过程和循环之间的并行性—功能并行性—的挖掘。并行编译的一个重要课题是提出一种中间表示形式,将源程序转换为中间表示形式,这样才能利用一些有力的工具如图论、代数等进行依赖分析。中间表示形式的优劣影响着分析的精确性和开销,并且影响着代码生成的质量。数据并行性的分析可以采用流图作为中间表示,但其粒度小且固定,对于大型程序分析代价过大,尤其不适用于功能并行性的分析。HTG 正是在这样的背景下提出的[2]。

NUAPC 是我们实现的一个基于 C^{++} 的自动并行编译分析工具[5,10,11],旨在分析 C^{++} 源程序的功能并行性,它主要分为流图分析,层次分析(HTG 化),控制依赖分析,数据依赖分析几部分。层次分析技术是 NUAPC 中分析的基础,它的工作是将源程序转换成层次式任务结构——层次任务图(HTG),实现中我们采用了改进的 HTG 作为自动并行编译的中间表示形式。

2 HTG 分析和控制依赖分析

2.1 定义

为便于展开讨论,我们先给出一些基本定义:

·基本块:基本块是指程序中一顺序执行的语句序列。它只有一个入口和一个出口,入口是指其中第一条语句,出口是指其中最后一条语句。对基本块而言,执行时只能由入口进入,从出口退出。

·控制流程图(CFG):是指具有唯一首结点的有向图。首结点是指从它开始到流图中任何结点都有一个通路的结点。控制流程图的结点是基本块,它的边集构成如下:设流图中结点 i 和结点 j 分别对应于程序中的基本块 i 和基本块 j,则当(i)基本块 j 在程序中的位置紧跟于基本块 i 之后,并且基本块 i 的出口不是转移语句和停语句。(ii)基本块 i 的出口是 goto(s)或 if…goto(s)。并且 s 是基本块 j 的入口语句。那么结点 i 和结点 j 之间存在一条边(i,j)。

维普资讯 http://www.cqvip.com

·必经关系：在 CFG 中，若结点 x 到 STOP 结点的任一条路径（不含 x）都包含 y，则 x 必经 y，记为 yΔpx，必经关系可以表示为一棵树，称为必经树，根为起始结点，每一个结点都必经它的子孙结点。

·回边：在 CFG 中，对于一条边(a,b)，若有 b 必经 a，那么称(a,b)为回边。回边在流图分析中有着重要的作用，对于归约图，每一回边代表着一个循环[1]。

将 HTG 用于功能并行性的分析主要是由 Girkar 提出的[2]，HTG 应用于并行编译中的一种程序的中间表示，是有层次的，层次划分是基于强连通区域，每一层次为一有向无环图，HTG 技术共划分四类结点：

·简单结点：对应于 CFG 的一个基本块，代表一个没有子任务的结点。

·复合结点：对应于 CFG 的一个过程调用块，是可以划分为更细的子任务的结点。

·循环结点：对应于 CFG 中的一个强连通区域即循环，它的循环体代表一个任务。它也可以划分为更细的子任务。

·特殊结点：有 START 结点和 STOP 结点。为了在并行代码中动态划分任务，必须在每个 HTG 层次的开始与结束加入特定的代码，这些代码就封装在 START 结点和 STOP 结点之中。

Girkar 的 HTG 中的层次思想源于编译优化中的层次结构分析技术[1]。但他认为文[1]中基于区间的层次分析技术并不适用于自动并行编译。因为：1）区间不是基于强连通区域的，就是说一个区间可能除了包含循环外，还包含该循环上挂下来的结点。2）区间分析技术对于非归约图要求应用结点分裂技术。于是，他提出了基于强连通区域的层次结构，层次结构的每一个结点都是 HTG 的一个任务。HTG 是一种优秀的程序的中间表示形式，首先是它的层次划分合理，每一层次逻辑上较其他的划分方法更接近于一个任务（并行单位），并且它的表示清晰，这一点在代码生成阶段能很好地利用。另外一点是它将每一层次无环化，无环化后的依赖分析得到的结果较直接对 CFG 进行分析要少。并不是因为那些依赖关系不存在，而是我们的功能并行性分析不需要考虑这些依赖关系。那些减少的依赖关系是 CFG 的回边造成的依赖，HTG 分析中不考虑循环的两次迭代造成的依赖，我们的模型中循环都是顺序执行的，对循环的并行化或优化可以放在功能并行分析的下一层次来完成，亦即可通过数据并行性的进一步分析来完成，而不需要在功能并行性分析中考虑。

可以看出，HTG 的任务控制较为灵活，基于 HTG 产生的并行代码可以动态地划分、组合任务，任务的粒度大小可以根据实际情况来调整。另外，由于 HTG 是有向无环图，在其之上进行控制依赖与数据依赖分析效率也较高，因此在自动并行编译中 HTG 有着很好的应用。

在利用 HTG 进行功能并行性分析时，我们发现 Girkar 的 HTG 也有其缺点，即每一层次的结点数很多而且大量是基本块结点，每一基本块只包含很少的语句（很少有超过6句的），对基本块进行功能并行性分析，没有很大的意义。将基本块作为一并行单位是不适合的，通信的代价可能比代码执行的代价要大。另外，结点多导致的直接问题是依赖分析的代价过大。并行编译的对象一般是大的程序，所以这一缺陷是致命的。

相比之下，区间的层次划分结点较少，我们试图将区间分析的优点引入 HTG 技术中，减小其结点数。直观地讲，是将每一层次的相邻接的基本结点块合并为一个大的结点，从而减少结点数并且不影响 HTG 的优点。图1（取自[2]）表示三种不同的层次划分的结果（仅画出第一层）。(A)是区间方法，(B)是 HTG 方法，(C)是我们采用的方法。可以看出(A)结点最少，有3

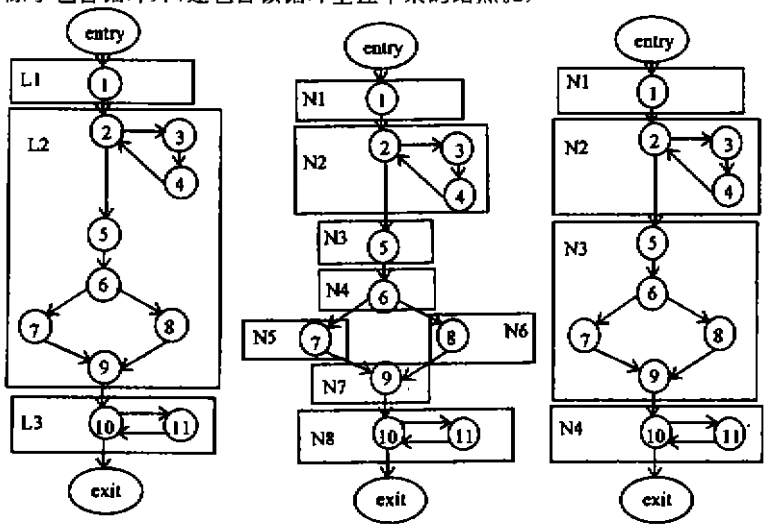


图 1

维普资讯 http://www.cqvip.com

个结点(L1至L3),但其结点不适合于表述功能并行性。(B)中有8个结点(N1至N8),相当多的结点是简单结点(N3至N7)。而(C)中有4个结点(N1至N4)。它集中了(A),(B)的优点即结点少与逻辑表示清晰。

2.2 HTG分析的过程

①结点的合并。每一HTG层次中的基本块结点对应于CFG的基本块结点,合并HTG的基本块结点可以通过合并CFG的基本块结点来实现。设简单基本块结点为除过程调用结点和回边的目的结点外的其他结点。我们设输入为CFG(V,E),起始结点为n_0,n为CFG中结点。定义CFG合并后为MFG(MV,ME),其中MV为V的幂集,设M(n)∈MV,则

·n在M(n)中;

·若m的所有前驱结点都是简单基本块结点,并且这些前驱结点都在M(n)中;

·没有其他的结点在M(n)中。

ME为E的子集,若有M(a)、M(b)∈MV,(M(a),M(b))∈ME iff 存在c∈M(a),d∈M(b),使得(c,d)∈E。

比较MFG和HTG可以看出MFG中有2类结点:①M(n)中只包含n;②M(n)中除包含n外,还包含其他结点。我们称后一种情况中的结点为合并结点。

要进行结点的合并,首先要对结点进行分类,区分出简单基本块结点,过程调用结点可以直接在流图中区别出来,而LOOP结点是和回边联系在一起的,所以首先要找出回边。将每个CFG的ENTRY作为头结点,对CFG作DFS(深度优先)分析,分析时碰到的边(x,y)可以归结为以下四类:

(1)结点Y没有被访问过,这时(x,y)为竖向边。

(2)有一个从y到x的由竖向边组成的路径P,这时(x,y)为回边。

(3)有一个从x到y的由竖向边组成的路径P,这时(x,y)为前边。

(4)否则,(x,y)为横边。

给定CFG产生回边的算法如下:

输入:流图CFG其首结点为n_0

输出:回边集合B

```
procedure search(n);
        begin
            标记n为已访问;
            for n的每个后继s do
              if s没有被访问 then search(s);
            dfn[n]:=i;
            i:=i-1;
            if dfn(n)≥dfn(s) then 将(n,s)加入B;
    end;
    begin
        B:=empty;{回边的集合}
        for CFG的每个结点n do 标记n为未被访问;
        i:=CFG的个数;
        search(n0);{n0为初始结点}
    end;
```

由CFG产生MFG的算法如下:

输入:流图CFG,其首结点为n_0

输出:MFG

```
procedure construct(M(n))
    begin
        M(n):={n};
        将n所有的后继改为M(n)的后继;
        while 存在m(m≠n0),且m的所有前驱结点都是
        简单基本块结点,并且这些前驱结点都在M(n)
        do
            begin
                M(n):=M(n)∪m;
                将m所有的后继改为M(n)的后继;
    end;
begin{主程序}
    construct(M(n0));
    选中n0;
    while 存在m,使得m未被选中但m的一个前驱被选
    中 do
        construct(M(m));
end
```

②循环的划分,HTG是基于循环的,HTG分析的第一步就是将循环从MFG之中划分出来。由一个回边划分循环的算法如下:

输入:流图MFG和回边集合T

输出:由T中回边确定的循环包含的结点构成集合的集合L

```
procedure insert(m);
    begin
        if m不属于loop then begin
            loop:=loop∪{m};
            将m压入栈;
    end;
begin
        for B中的每个元素(n,d)do
        begin
            loop:={d};
            insert(n);
            while 栈非空 do begin
                弹出栈顶m;
                for m的每个前驱P do insert(P);
            end
            将loop加入L;
        end
end
```

③生成HTG结点集

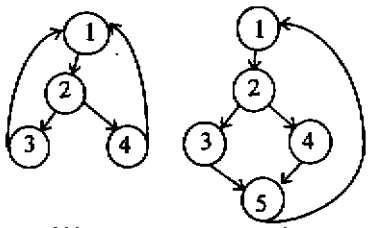
Ⅰ.*循环的合并*。对于归约图,设L(a,b)与L(c,d)是2个循环,则若B≠D则$L(a,b)\cap L(c,d)=\varnothing$或$L(a,b)\subset L(c,d)$或$L(c,d)\subset L(a,b)$。这个性质是MDG层次化的基础。对于B=D,可能出现图2(A)的情况,这时无法分清两个循环的包含关系。解决办法是将两个循环合并为一个,图2(A)可以转换为图2(B)。这就是循环合并所做的工作。其描述如下:令$S_x=\{y;y\in V.\exists\ (y,x)\in L\}$,$I(x)=\bigcup_{r\in S_x} L(y,x)$。

Ⅱ.*HTG结点集*。循环合并后,生成的I(x)为强连通区域,HTG的结点集如下:

维普资讯 http://www.cqvip.com

$HV=\{I(x):x\in H(G)\}\cup\{V\}\cup V\cup$(附加的结点)

附加的结点就是前面所提出的 START 结点和 STOP 结点。



(A) 图2 (B)

④HTG 边集的产生，对于任两个 HTG 结点 A 与 B，$A\cap B=${A 或 B 或 $\varnothing$}。定义 $P(A)=\{B;B\in HV, A\subset B\}$，每个$(P(A),\subset)$是一个全序关系，所以我们可以定义 $P_{min}(A)$：令 $P_{min}(V)=$NULL，则 P_{min} 是 HV 上的函数，为方便起见，设 $f=P_{min}$；每个循环结点和合成结点，都包含着其它的子任务，在 HTG 中表示为每个这样的结点对应于一个子图。

在给出 HTG 边集的产生算法之前，讨论一下 START 结点，对于复合结点，当控制到达 STOP 结点时，任务已经完成，可以返回上一层 HTG；对于循环结点，执行 STOP 结点时，循环体可能要再执行，也有可能结束。所以在 STOP 结点中必须有两个域分别记录结束条件和在执行条件。

构造 HTG 边集的方法如下：

设结点 a 对应的 HTG 结点为 g(a)。对 CFG 的每个边(a,b)，应用如下的规则：

1. 找到一个 z，使得 $z=f^n(a)=f^m(b)$，这里 $n,m\geq 1$，并且 m，n 使得最小。

2. FOR $0\leq i\leq n-2$，加边$(g(f^i(a)),STOP(g(f^{i+1}(a))))$到 $HE(g(f^i(a)))$，并在 $STOP(g(f^{i-1}(a)))$结点中加入结束条件 c(a,b)。

3. FOR $0\leq i\leq n-1$，加边$(START(g(f^i(b)),g(f^{i-1}(b))))$到 $HE(g(f^i(a)))$。

4. 若(a,b)不是回边，加边$(g(f^{n-1}(a)),g(f^{m-1}(b)))$到 HE(g(z))。

5. 若(a,b)是回边，加边$(g(f^{n-1}(a)),STOP(g(a)))$和$(START(g(z)),g(f^{m-1}(b)))$到 HE(g(z))并在 $STOP(g(f^{i-1}(a)))$结点中加入重新执行条件 c(a,b)。

6. 在每一层 HE(g(a))中加入边(STOP(g(a)),g(a))到 HE(g(a))。

小结 依赖关系分析的基础是源程序的流图分析，我们的流图分析使用 LEX 与 YACC 来实现，对过程调用，我们做了特殊的处理：对于系统提供的函数与过程，我们做串行化处理，即将它等同于一般的顺序语句；对程序自定义的函数与过程我们将该调用所在的语句单独作为一基本块，称之为过程调用块。这样做是因为对程序自定义的函数与过程可以精确地分析其依赖关系，减小过程调用块的大小无疑会减少该块与其它块的依赖，从而提高过程之间的并行性。所用 C++ 语法为 James A. Roskind 的 version 1.0，我们对其中不适合的部分做了修改或扩充，如对语法中过多的类型描述进行了约减，对控制语句的描述进行了扩充。

我们在 SUN SPARC 2工作站和 Wyse series 7000i 并行机上，实现了上述分析过程。Wyse series 7000I 是一有四个80486芯片的对称多处理器并行机，操作系统是 SCO UNIX。但现在的 HTG 分析算法较复杂，有待于以后的工作来改进。

参考文献

[1] V. Aho et al., Compilers: Principles Techniques and Tools, Addison Wesley, 1986

[2] M. B. Girkar, Functional Parallelism Therotical Foundations and Implementations, Ph. D. dissertation, University of Illinois, 1991

[3] J. Freeante et al., The Program Dependence Graph and Its Use in Optimization, ACM Trans. on Language and Systems, 9, 1987

[4] K. Pingali, G. Bilardi, APT: A Data Structure for Optimal Control Dependence Computation, ACM Sigplan Notice, 6, 1995

[5] Xie Li et al., A Model for Automatically Parallelizing Compiler Based on C++, Proc. of 2nd Intl. Computer Science Conf., Hong Kong, 1992.12

[6] P. Carlin et al., The Compositional C++ language definition, CS TR-92-02, California Institute of technology, 1992

[7] F. Bodin et al., Implementing a Parallel C++ Runtime System for Scaleable Parallel Systems, Proc. 1993 Supercomputing Conf., Portland, Nov. 1993

[8] D. Gannon and J. K. Lee, Object Oriented Parallelism: PC++ Ideas and Experiment, Proc. Japan Society of Parallel Processing 1991

[9] F. Bodin et al., Implementing a Parallel C++ Runtime System for Scaleable Parallel Systems, Proc. Supercomputing 93, 1993

[10] Zhu Genjiang et al., A path-based method of parallelizing C++ programs, ACM Sigplan notices, 29(2) 1994

[11] Zhu Genjiang et al., A new methodology of data dependence analysis for parallelizing C++, ACM Sigplan Notice 30(12)1995