软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(2):411−427 [doi: 10.3724/SP.J.1001.2012.03919]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种利用并行复算实现的 OpenMP 容错机制*

富弘毅[1+], 丁 滟[2], 宋 伟[1], 杨学军[1]

[1](国防科学技术大学 并行与分布处理国防科技重点实验室,湖南 长沙 410073)

[2](国防科学技术大学 计算机学院 软件研究所,湖南 长沙 410073)

Fault Tolerance Scheme Using Parallel Recomputing for OpenMP Programs

FU Hong-Yi[1+], DING Yan[2], SONG Wei[1], YANG Xue-Jun[1]

[1](Key Laboratory of Science and Technology for National Defense of Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China)

[2](Institution of Software, College of Computer, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: E-mail: fool_20022004@yahoo.com.cn

Fu HY, Ding Y, Song W, Yang XJ. Fault tolerance scheme using parallel recomputing for OpenMP programs. *Journal of Software*, 2012,23(2):411−427. http://www.jos.org.cn/1000-9825/3919.htm

Abstract: This paper proposes a fault tolerance approach for OpenMP programs, named PR-OMP, which makes use of a novel fault recovery scheme, parallel recomputing. By redistributing the workload of the failed thread to all the surviving threads, PR-OMP remarkably reduces the overhead for fault recovery. The paper discusses the key issues including program division, computational state saving, workload redistribution, and fault detection of PR-OMP and details concerning implementation. Furthermore, the paper also presents an extended data flow analysis for OpenMP, which is used to decrease the data amount of computational state saving. Through the experimental evaluation, it has been proven that this approach achieves a minor overhead in fault recovery.

Key words: fault tolerance; OpenMP; parallel recomputing; data-flow analysis

摘 要: 基于并行复算的故障恢复技术,将故障恢复的计算任务分配至未发生故障的结点上并行执行,从而显著缩短复算时间,有效降低故障恢复开销,提高并行程序容错性能.基于该故障恢复技术,提出了一种针对 OpenMP 并行程序的容错机制 PR-OMP,有效解决了分段复算、复算负载重分布等问题;此外,还扩展了传统编译数据流分析技术,提出了针对 OpenMP 并行程序的数据流分析技术,并基于该技术计算状态保存开销进行优化.设计实现了用于支持 PR-OMP 的编译工具 GiFT-OMP,并通过实验证明了 PR-OMP 机制及其支持工具的有效性,评估并分析了其性能和可扩展性.

关键词: 容错;OpenMP;并行复算;数据流分析

中图法分类号: TP316 **文献标识码**: A

近年来,大规模并行计算机系统的规模和计算能力均有大幅度的提高,根据 2010 年 6 月公布的高性能计算机系统 Top 500 列表[1],目前已有 7 台超级计算机系统处理器/处理器核的个数超过了 10 万个,其中 3 台甚至达

* 基金项目: 国家自然科学基金(60921062, 61003087); 国家高技术研究发展计划(863)(2009AA01Z102)
收稿时间: 2010-01-05; 修改时间: 2010-03-30; 定稿时间: 2010-07-06

到 20 万个.硬件的复杂性使系统可靠性问题变得越来越严重,高性能计算机系统的平均无故障时间(mean time between failure,简称 MTBF)和以前相比大幅度下降,甚至达到若干个小时的量级.例如,Google Cluster 大约每隔 36 小时就会出现结点失效,而 ASCI White 系统的 MTBF 约为 40 个小时左右[2].另一方面,大规模科学计算应用程序的数据规模、计算复杂性和运行时间仍维持较高的水平,例如,Blue Gene 上的 protein-folding 程序的运行时间长达数月.因此,要解决硬件平台低可靠性与应用大规模长时间运行之间的矛盾,就必须考虑采用适当的容错技术,尽可能地提高系统可靠性,以满足应用的需要.传统的基于共享存储体系结构的并行计算机系统规模有限,因此针对共享存储系统的容错技术一度未受关注.但随着大规模 NUMA 系统以及基于 SMP 结点构建的大规模 MPP 系统在高性能计算领域日益广泛的应用,针对共享存储体系结构的容错技术研究日益重要.

OpenMP 编程模型在共享存储系统中被广泛地用于开发并行性,而目前针对 OpenMP 程序的回滚-恢复容错机制研究还不多.早期的研究主要致力于提供系统级的解决方案.例如,SaftyNet[3]和 ReVive[4]借助专用的硬件记录共享存储系统中本地结点存储单元内容的变化以及结点间的消息,以支持系统状态的保存和恢复.但系统硬件依赖度高,难以在一般科学计算应用中推广;文献[5]是一种软件实现的方案,扩展了共享存储一致性协议,定位共享页面在系统中的位置,以便能够正确地保存全局计算状态.系统级解决方案的缺陷在于其紧密的平台相关性,不利于移植和扩展.针对这一问题,Greg 等人提出了编译辅助的 OpenMP 应用级检查点机制[6,7],并集成在 C^3[8]容错编译框架中.C^3 系统可以根据用户指定的编译指导命令以源到源的方式将普通的 OpenMP 程序转换成具有应用级检查点机制的容错程序,具有维护全局状态一致性的能力.应用级方案具有很好的平台无关性,移植性好,但在容错性能方面存在两个问题:一方面,在程序的正常执行过程中,周期性的检查点保存引入了一定的 I/O 操作开销,对于数据规模较大的程序,这部分开销将极大地增加程序的总执行时间;另一方面,在故障恢复的过程中,所有的线程都要回滚到最近一次检查点,即使没有发生故障的线程也需要重复执行,造成计算资源的极大浪费.这两个问题是提高容错技术性能的关键.

在先前的研究工作中,我们提出了面向 MPI 消息传递并行程序的基于并行复算的快速故障恢复方法——容错并行算法(fault tolerant parallel algorithm,简称 FTPA)[9].其基本思想是,在程序的执行过程中定期地进行计算状态保存,在有故障发生时回滚到最近保存过的计算状态,然后利用未发生故障的 MPI 进程并行地完成故障恢复的计算.FTPA 的提出,旨在从降低计算状态保存开销和故障恢复开销的角度,尽可能地提高容错性能.本文针对 OpenMP 并行程序进行基于并行复算的容错技术研究,首先,基于 OpenMP 并行机制对 OpenMP 程序进行适当的分段,有效减小了每次故障恢复的计算量;其次,通过对故障恢复过程的并行化执行,显著缩短了故障恢复时间,从而有效降低故障恢复开销;最后,提出针对 OpenMP 并行程序的数据流分析技术,用于减少计算状态保存的数据量,有效地降低计算状态保存开销.

1 基于并行复算的容错机制

1.1 复算及相关概念的定义

程序的执行是通过执行一系列语句对一组变量施加的一系列操作,而每个操作都会修改某个变量的值.

定义 1. 令程序的变量集合为 $V=\{v_1,v_2,...,v_M\}$,语句集合为 $S=\{s_1,s_2,...,s_N\}$,称程序运行至语句 s_i 之前时 V 中所有变量的值为语句 s_i 之前的应用级计算状态,简称为计算状态,记为 C_i.

程序的流图定义了 S 中的语句被执行的顺序,若语句 s_i 先于语句 s_j 执行,则在流图中,s_j 是 s_i 的后继.进一步地,如果 s_j 是 s_i 的后继,且对于任一 s_k,s_k 是 s_i 的直接后继当且仅当 s_k 是 s_j 的后继,或 $s_k=s_j$.若在流图中语句 s_i 的直接后继是语句 s_j,则执行语句 s_i 将导致计算状态从 C_i 迁移到 C_j.

定义 2. 在程序运行过程中的某一时刻,将当时的计算状态 C_i 记录到存储介质上,称为保存计算状态 C_i;在程序运行过程中的某一时刻,将 V 中所有变量的值设置为先前保存过的计算状态 C_i 中的值,称为恢复计算状态 C_i,亦称为将计算状态回滚到 C_i.

定义 3. 若在程序流图中语句 s_j 是 s_i 的后继,则从 s_i 到 s_j 的所有路径上包含的除 s_j 外的所有语句集合称为一个程序段,记为 $[i,j)$.

定义 4. 在程序的执行过程中,在执行到语句 s_j 之前时检测到计算状态 C_j 错误的情况下,通过将计算状态回滚到 C_i,重复执行程序段$[i,j)$,使计算状态 C_j 恢复正确的容错方法,称为对该程序段进行的复算.

1.2 并行复算的基本思想

并行程序的任一程序段$[i,j)$在由 k 个任务并行执行时(本文中的"任务"意指参与执行并行程序的实体,例如 MPI 进程或 OpenMP 线程),任一任务 $T_r(1\leqslant r\leqslant k)$均执行$[i,j)$的一个副本,记为$[i,j)_r$.通过这种方式,$[i,j)$中包含的计算被划分给 k 个任务,而$[i,j)_r$中包含分配给任务 $T_r(1\leqslant r\leqslant k)$的计算.

定义 5. 设并行程序由 k 个任务 $T_1,T_2,...,T_k$ 并行执行,当任务 T_r 执行到语句 s_j 之前时,称程序变量集合 V 中所有被 T_r 所访问的变量的值为 T_r 上 s_j 处的局部应用级计算状态,简称为局部计算状态,记为 C_j^r.

并行程序在语句 s_j 之前的计算状态 C_j 是一个 k 维的计算状态向量,其中,每个元素对应一个任务在语句 s_j 之前的局部计算状态,即 $C_j=\langle C_j^1,C_j^2,...,C_j^k\rangle$.

在并行程序的执行过程中,如果任务 T_r在执行到语句s_j之前检测到局部计算状态 C_j^r 错误,那么,如果任务之间不存在数据相关,则只需将任务 T_r的计算状态回滚到先前保存的某个计算状态 C_i^r ,然后重新执行 T_r的程序段副本$[i,j]_r$,即可实现 C_j^r 的正确恢复.并且,由于不存在数据相关,所有无故障的任务都不需要对自己的程序段副本进行复算,而是在故障任务,即 T_r 进行复算期间空闲等待.针对这一特性,我们提出了基于并行复算的容错机制.

定义 6. 基于并行复算的容错机制(parallel-recomputing-based fault tolerance,简称 PR-FT)是指,在将一个普通的并行程序按照某种原则划分成若干个程序段的基础上添加适当的代码成分,使其能够在运行时在每个程序段的入口处完成计算状态的保存,在出口处完成故障检测,并在某个任务上检测到故障的情况下,通过使用无故障的任务重新并行地执行故障任务的程序段副本来加速故障恢复的容错方案.

采用多个任务进行故障恢复,增加了该过程中再次发生故障的概率.因此,在并行复算结束时也应该进行故障检测.如果的确有故障发生,则只需重新恢复计算状态,再次进行并行复算即可.为简化描述,在接下来的讨论中假设故障恢复过程中不再发生新的故障.

设一个普通的并行程序被划分为 m 个程序段,$F_1,F_2,...,F_m$,那么,将该程序改造成 PR-FT 程序需要对其中的每个程序段 $F_i=[i,j)$进行改造如下:

- 在语句 s_i 前添加计算状态保存的代码,称为状态保存段,记为 SS_i;
- 在语句 s_j 之后添加故障检测的代码,称为故障检测段,记为 ED_i;
- 在故障检测段之后添加计算状态恢复的代码,并基于 F_i 构造对 F_i 进行并行复算的代码,这两部分代码合称为故障恢复段,记为 ER_i.

SS_i,F_i,ED_i 和 ER_i 共同构成了 PR-FT 程序的第 i 个并行复算段,记为 PR_i.

综上所述,一个 PR-FT 程序的结构如图 1 所示.

按照 PR-FT 的基本思想,针对具体的并行程序实现 PR-FT 机制主要涉及以下两个关键问题:

- 程序段划分问题:程序段划分确定了故障检测的间隔和一次并行复算所要处理的计算负载,是实现 PR-FT 的首要问题.
- 复算负载再划分问题:并行复算是利用无故障任务重复处理故障任务上的程序段副本所包含的计算负载的过程,因此,如何确定故障任务上的负载,将其并行划分并调度到无故障任务上执行,是决定基于并行复算的 OpenMP 程序容错机制性能的关键问题.

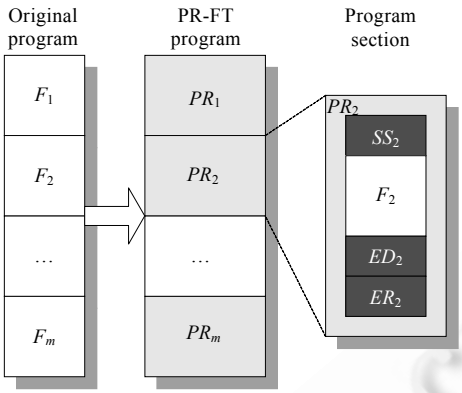


Fig.1 Structure of a PR-FT program

图 1 PR-FT 程序结构

1.3 可并行复算的条件

设并行程序的某一程序段 F_i 的计算负载为 W_i,当 F_i 由 k 个线程执行时,W_i 被以某种方式分解为可并行执行的 k 个部分,$W_i^1,W_i^2,...,W_i^k$. 若无论划分方式如何,任一 W_i^k ($1\leqslant r\leqslant k$)都可再次分解为可并行执行的 m 个部分,则称 F_i 是可再划分的,那么对于并行程序中的一个程序段 F_i,F_i 或其中包含的一个程序段 $\tilde{F}_i$ 是可再划分的,是 F_i 可并行复算的一个必要条件.

进一步地,若程序段 F_i 中包含一段可再划分的程序段 $\tilde{F}_i$,其计算负载为 $\tilde{W}_i$,那么在对 F_i 进行并行复算时,$\tilde{F}$ 在某个线程 T_r 上的副本所包含的负载 $\tilde{W}_i^r$ 将被再划分,因此,必须能够确切地知道 $\tilde{W}_i^r$ 中具体包含哪些计算才能完成再划分.而 $\tilde{W}_i^r$ 所包含的内容可能是静态分析可知的,也可能是由程序的执行过程动态确定的.但无论怎样,都必须能够在并行复算之前通过某种方式确定下来,这是 F_i 可并行复算的另一个必要条件.

在具体的应用程序中实现并行复算机制时,可以根据上述两个条件判断程序段中哪些成分是可以并行复算的.

2 针对 OpenMP 程序设计并行复算容错机制

本节提出针对 OpenMP 并行程序实现的 PR-FT 机制,称为 PR-OMP.PR-OMP 通过源到源转换的方式将普通的 OpenMP 程序转换为 PR-FT OpenMP 程序,可以通过编译时静态分析和代码插装来实现.首先,面向 OpenMP 并行执行模型,对程序段划分问题和程序段副本计算负载再划分问题进行研究,提出简明、高效的解决方法;然后,对计算状态保存的方法进行研究,提出了基于变量活跃性分析的计算状态保存技术,对状态保存的性能进行优化;最后,讨论了 PR-OMP 所适用的故障模型以及相应的故障检测技术.

2.1 程序段划分

2.1.1 划分约束

程序段的划分是确定 PR-FT 程序结构的基础,一个较好的划分方案应该能够在保证 PR-FT 机制的有效性前提下,尽可能地有助于提高 PR-FT 的容错性能.因此,本节归纳了程序段划分的一些约束条件如下:

(1) MTBF 约束:MTBF 是指系统的平均无故障时间,是系统的固有属性.由于 PR-FT 程序每两次故障检测的间隔至少是一个程序段的运行时间,因此,如果程序段过长而导致其执行时间超过 MTBF,则在两次故障检测之间发生故障的概率会相当高,失去了检错的意义.因此,在分段时应注意程序段的长度不宜过长.

(2) 容错性能约束:在每个程序段的前后都要进行计算状态保存和故障检测,如果程序段数量太多就会导

致这两部分工作的开销过高,从而严重影响 PR-FT 程序的性能.因此,程序段的长度也不宜太短.

(3) 可并行化约束:程序段在每个线程上的副本是并行复算的对象,因此,其中所包含的负载应该是能够被进一步并行划分的,这在程序段划分过程中也是应当考虑的因素.

(4) 语言机制约束:并行程序的语言机制也会对分段的方法产生限制.例如,一个程序段不能跨越 OpenMP 并行区这样的程序结构的边界,等等.

在上述 4 个约束条件中,约束 1 和约束 4 关系到并行复算机制的有效性,是强制性约束,即必须被满足的约束条件.这是因为,不满足约束 1 会导致并行复算机制完全失效,而不满足约束 4 则导致并行复算机制无法针对具体程序而实现.约束 2 和约束 3 关系到并行复算机制的容错性能,是非强制性约束,应在在保证满足约束 1 和约束 4 的前提下尽量满足.

2.1.2 初始划分方案

OpenMP 具有显式定义的并行区域,令 S_i 和 P_i 分别表示一个 OpenMP 程序中的第 i 个串行区和第 i 个并行区,则该 OpenMP 程序可以表示为由串行区和并行区交替构成的序列 $S_1P_1S_2P_2...S_nP_nS_{n+1}$,其中,$S_i(1\leqslant i\leqslant n+1)$可能为空.

根据约束 1 和约束 2,程序段中包含的计算量不应太多,但也不应太少;根据约束 3,程序段在每个线程上的负载都应该能够被并行化;此外,约束 4 又要求程序段应该以某个 OpenMP 并行结构为边界.综合考虑这些因素,在初始划分时,以 OpenMP 并行循环作为程序段划分的边界,即每个程序段中最多包含一个并行循环.选择并行循环作为程序段划分的边界具有如下的优势:

- 并行循环的计算量较大;
- 循环迭代间不存在数据相关;
- 在 OpenMP 程序中显式地描述.

根据 OpenMP 并行区中所包含的并行循环个数,可将其划分为两类:

- 简单并行区:称最多包含 1 个并行循环的并行区为简单并行区;
- 长并行区:称包含多个并行循环的并行区为长并行区;
- 程序中任一简单并行区 P_i 中最多只包含 1 个并行循环 L_i,若 P_i 和 P_j 是两个简单并行区,且对于所有 P_k,$i<k<j$,P_k 中不包含并行循环,即定义 $S_{i+1}P_{i+1}S_{i+2}P_{i+2}...S_jP_j$ 为一个程序段.

如果存在长并行区 P_i,其中包含 m 个并行循环 $L_i^1,L_i^2,...,L_i^m$,那么,可以在进行程序段划分之前先对 P_i 进行预处理,将其 P_i 分割为一组具有等价语义的简单并行区 $P_i^1,P_i^2,...,P_i^m$,称为长并行区 P_i 的生成并行区组.在进行程序段划分之前,首先将序列 $S_1P_1S_2P_2...S_nP_nS_{n+1}$ 中所有的长并行区替换为其生成并行区组,然后再对该序列进行划分.

为了保证分割的正确性,必须保证所有的变量在长并行区及对应的生成并行区组中具有等价的数据作用域属性.由于篇幅所限,在这里仅以图的形式给出变量的数据作用域属性从长并行区到其生成并行区组的转换规则,如图 2 所示.

按照上述划分方案,每个程序段中除了包含并行循环除外,还可能包含串行执行部分,最后一个程序段除外,它只包含串行代码.图 3 给出了一个程序段初始划分的示意图.

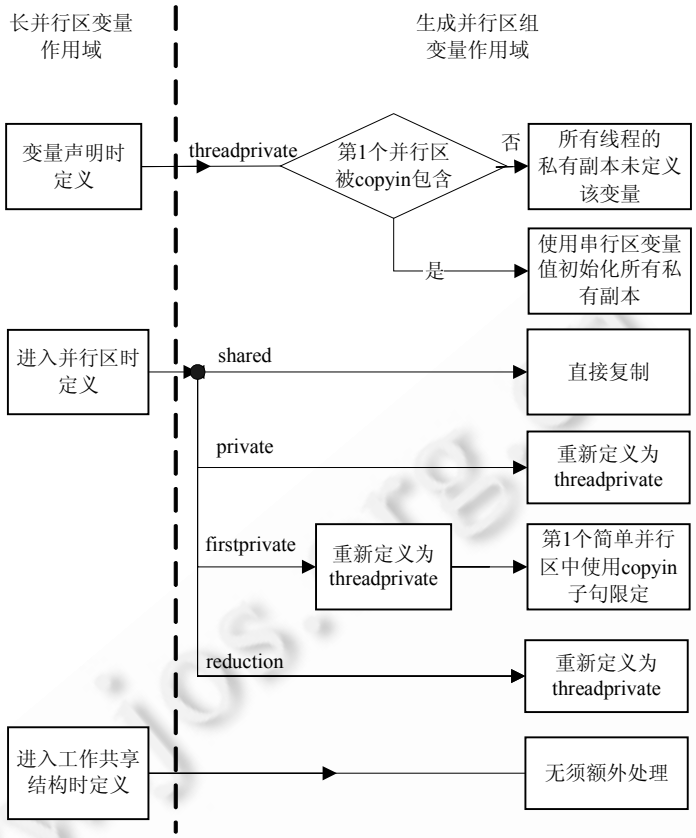


Fig.2 Data scope attributes transformation rules

图 2 数据作用域属性转换规则

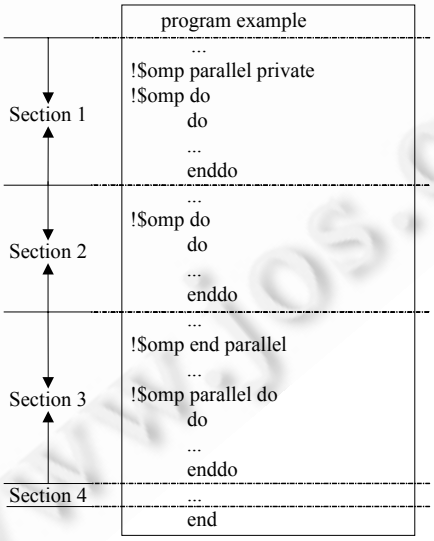


Fig.3 A sample of division

图 3 程序段初始划分示例

2.1.3 基于用户编译指导的划分修正

由于程序中可能存在运行时间极长的并行循环,这样,按照先前讨论的程序段初始划分结果可能仍然不满足约束 1 的要求.在这种情况下,需要对初始划分结果进行修正,即对运行时间较长的程序段进行分割.由于程序段的长短是一个运行时的概念,在编译时难以准确地度量,因此,这部分信息将由用户指定.用户可以用编译指导语句的形式指定某个并行循环应该被分割成几个程序段,形式如下:

!$OMP DO *PR_FRAGS*(*num_prs*),

其中,*PR_FRAGS* 是 PR-OMP 所定义的 OpenMP 指导语句子句,参数 *num_prs* 指定了该并行循环应该被划分到 *num_prs* 个程序段中.

按照程序段初始划分方案,每个程序段的最后都包含一个并行循环.对于程序段 F_i,令其中包含的并行循环为 L_i,其余部分为 R_i,则 $F_i=R_iL_i$.若 L_i 上指定了子句 *PR_FRAGS*(*m*),则首先将 L_i 分割成 m 个循环 $L_i^1,L_i^2,...,L_i^m$, 再将 F_i 分裂成 m 个程序段 $F_i^1,F_i^2,...,F_i^m$, 其中, $F_i^1=R_iL_i^1, F_i^r=L_i^r$ $(2\leqslant r\leqslant m)$.

循环 L_i 的分割是对其迭代集合的划分,若其迭代集合 $IR_i=[1,N]$,则 L_i^r 的迭代集合为

$$\left[(r-1)\left\lceil\frac{N}{m}\right\rceil+1,\min\left\{r\left\lceil\frac{N}{m}\right\rceil,N\right\}\right],\quad 1\leqslant r\leqslant m.$$

2.2 程序段的并行复算

对于 OpenMP 程序中的任一程序段 $F_i=S_iP_i,S_i$ 显然是不可并行复算的.因此,需要在 P_i 中找出可并行复算的部分.在 OpenMP 的并行区中,可能包含 4 种任务共享结构,OMP DO,OMP WORKSHARE,OMP SECTIONS 和 OMP SINGLE.根据第 1.3 节给出的可并行复算条件,在对程序段 F_i 进行故障恢复时,只有其中的并行循环可以被并行复算.

设并行循环 L_i 的迭代集为 IR_i,由 k 个线程执行.根据 OpenMP 并行机制,IR_i 被分解为 k 个子集, $IR_i^1,IR_i^2,...,IR_i^k$, 任一线程 T_r 执行其中一个迭代子集 IR_i^r, 那么,若 T_r 在执行过程中发生故障,则 IR_i^r 为并行复算的负载.对 L_i 进行并行复算即是对 IR_i^r 进行再划分,并分配给所有的无故障线程执行,其中包含两个方面的问题:

- 确定并行复算的负载:即确定 IR_i^r 中包含哪些迭代;
- 确定并行复算的调度方式:以何种方式将 IR_i^r 中的迭代分配给无故障线程.

2.2.1 确定并行复算的负载

对于并行循环 L_i 以及某个参与执行的线程 T_r,要确定该线程被分配的迭代子集 IR_i^r, 必须考虑 L_i 所使用的 OpenMP 并行循环调度策略,包括静态调度(STATIC)策略和动态调度(DYNAMIC)策略.

(1) 静态调度策略

静态调度 OpenMP 并行循环具有如下形式:

```
!$OMP PARALLEL DO SCHEDULE(STATIC,chunk)
      DO i=L,U,S
        H(i)
      ENDDO
```

该并行循环可以抽象为一个四元组⟨*L*,*U*,*S*,*chunk*⟩,其中,*L*,*U*,*S* 分别表示循环迭代索引 *i* 的下界、上界和步长,*chunk* 表示调度单元的大小.根据 *L*,*U*,*S*,还可以导出迭代数 *N* 和迭代号 *I*,其关系如下:

$$i=L+I\times S,$$
$$N=(U-L+1)/S.$$

若 L_i 是一个静态调度的并行循环,那么其迭代集合 IR_i 中任一迭代的迭代号 I 和该迭代所在的循环块的块号 b 之间存在如下关系:

$$b=\lfloor I/chunk\rfloor \tag{1}$$

若 L_i 由 k 个线程执行,那么对于某个线程 $T_r(1\leqslant r\leqslant k)$,分配给 T_r 的任一循环块的块号 b 与线程号 r 存在如

下的线性关系:

$$r=b \bmod k \tag{2}$$

根据公式(1)和公式(2),可以建立 IR_i 中任一迭代的迭代号 I 执行该迭代的线程的线程号 r 之间的关系,如下:

$$r=\lfloor I/chunk \rfloor \bmod k \tag{3}$$

利用公式(3)可以计算出在静态调度并行循环的执行过程中,任一线程上被分配的迭代子集,即并行复算过程所要遍历的迭代子集.

(2) 动态调度策略

动态调度 OpenMP 并行循环具有如下形式:

```
!$OMP PARALLEL DO SCHEDULE(DYNAMIC,chunk)
      DO i=L,U,S
        H(i)
      ENDDO
```

动态调度策略与静态调度策略仅在循环块的分配方式上有所不同.如果 L_i 是一个动态调度的并行循环,那么在该循环被分块后,各个循环块并不是在编译时就以确定的方式分配给各个线程,而是在运行时由所有线程异步地取来执行.即,每个线程在执行完一个循环块之后,立即取下一个迭代块来执行.因此,动态调度也称为自调度.

对于执行 L_i 的任一线程 T_r 来说,哪些迭代被分配到该线程上执行完全是不确定的,而是由执行过程中各个线程的相对执行速度、系统资源的竞争程度等因素决定.因此,无法确切地计算出某个线程被分配的迭代子集.

为了对动态调度的并行循环进行并行复算,可以在程序的正常执行过程中记录每个线程已经执行过的循环块,这一信息称为并行循环的调度簿记.与周期性的状态保存类似,每个线程都保存自己的调度簿记,并在每执行完一个循环块时更新调度簿记.这样,当某个线程发生故障时,只要根据该线程的调度簿记就可以确定需要复算的循环块集合.

调度簿记可以使用逻辑数组实现,其优点是空间开销比较低.对于 10 000 000 次这样规模的循环,需要的逻辑数组占用存储空间约为 1MB 左右.

2.2.2 确定并行复算的调度方式

对一个迭代子集进行划分并调度到无故障线程上去执行,实际上也是一个并行循环的执行过程,可以选择使用静态调度策略或动态调度策略.一般来说,静态调度完全在编译时实现,没有运行时开销,但不能保证负载平衡;而动态调度可以实现较好的负载平衡,但自调度会对性能造成影响.具体使用哪种策略,应该根据应用的特点来决定.

因此,PR-OMP 机制引入另一个扩充的指导语句子句,将并行复算的调度策略交给用户来选择,形式如下:

!$OMP DO *PR_SCHE*(*type*),

其中,参数 *type* 的取值可以是 *STATIC* 或 *DYNAMIC*,后者为缺省值.

2.2.3 并行复算代码的生成方法

对于某一程序段 F_i,其并行复算代码是基于 F_i 中的并行循环 L_i 生成的一个并行循环,并满足如下要求:

- 遍历 L_i 在一个线程上的迭代子集;
- 使用由子句 *PR_SCHE* 指定的调度策略.

根据第2.2.1节的讨论,并行复算代码所遍历的迭代子集取决于 L_i 所使用的调度策略.因此,并行复算代码的形式也因调度策略不同而不同.

任何一个并行循环⟨*L*,*U*,*S*,*chunk*⟩都可以被规范化,使下界为 0,步长为 1,上界为 *N*–1,即⟨0,*N*–1,1,*chunk*⟩.因此,本节只讨论规范化循环.

设 L_i 是一个具有如下形式的静态调度并行循环:

!$OMP PARALLEL DO *SCHEDULE*(*STATIC*,*chunk*)

```
DO i=1,N–1
  H(i)
ENDDO
```

该循环可变换为等价的二重循环,外层循环遍历循环块集合,内层循环遍历循环块内迭代集.而并行化针对外层循环展开,且 *chunk* 为 1.

设迭代数为 N,则迭代集 $IR=\{1,2,\ldots,N\}$ 被分割成$\lceil N/chunk\rceil$个循环块,第 b 个循环块起始于迭代号 $chunk\times b+1$,终止于迭代 $\min\{chunk\times(b+1),N-1\}$,则变换后的二重循环如下:

```
!$OMP PARALLEL DO SCHEDULE(STATIC,1)
DO b=1,⌈N/chunk⌉,1
  DO i=chunk×b+1,min{chunk×(b+1),N–1}
    H(i)
  ENDDO
ENDDO
```

在上述变换后的循环中,外层迭代空间遍历所有调度单元的集合.因此,可以根据任一线程的线程号确定分配给该线程的循环块集合,并相应地重新计算并修改上述循环中外层循环的下界和步长,即可将其转换为并行复算代码.

如果用户在 L_i 上指定了子句 *PR_SCHE*(*STATIC*),则并行复算代码如下:

```
!$OMP PARALLEL DO SCHEDULE(STATIC,1)
DO b=j,⌈N/chunk⌉,k
  DO i=chunk×b+1,min{chunk×(b+1),N–1}
    H(i)
  ENDDO
ENDDO
```

若 L_i 是动态调度的并行循环,那么首先还是将其转换为二重循环,只是由于在动态调度的过程中,一个线程上所执行的循环块的块号可能是完全没有规律可循的,不能像处理静态调度并行循环时那样通过修改上下界和步长来生成并行复算代码.因此,针对动态调度并行循环,其并行复算代码的上下界和步长均不变,其外层循环仍然遍历原循环的循环块集合,只是在每次进入迭代时访问故障线程的调度簿记来决定是否要执行本次迭代.并且,由于这种策略导致并行复算代码中循环迭代的执行时间不同,因此,应该采用动态调度的策略来进行并行复算.综上所述,若故障线程为 T_j,则动态调度并行循环的并行复算代码具有如下形式:

```
!$OMP PARALLEL DO SCHEDULE(DYNAMIC,1)
DO b=1,⌈N/chunk⌉,1
  if (skip(b,bk)) then
    DO i=chunk×b+1,min(chunk×(b+1),N–1)
      H(i)
    ENDDO
  endif
ENDDO
```

其中,*skip*(*b*,*bk*)根据当前块号 b 和 T_j 的调度簿记 *bk* 判断循环块 b 是否被分配给 T_j,即并行复算过程中是否要执行循环块 b.

2.3 优化的计算状态保存

2.3.1 计算状态保存可优化依据

并行复算的第 1 步是计算状态的回滚,即恢复最近保存过的那个计算状态.这就要求在每个程序段的入口

处,即每次故障检测之后保存计算状态.将程序空间中,所有变量全部保存下来是一种最基本的方法.但对于大多数程序来说,只要保存一部分变量就可以恢复计算状态,有的程序甚至只需要保存极少的几个变量就可以实现计算状态的恢复.考虑如图 4 所示的 OpneMP 矩阵乘程序.

```
1       program omp_mm
2       implicit none
3       integer a(4,4),b(4,4),c(4,4)
4       integer i,j,k
5       a=...
6       b=...
7       !saving computational state
8   !$omp parallel private(i,j,k)
9   !$omp do
10        do i=1,4
11          do j=1,4
12            do k=1,4
13              c(j,i)=a(k,i)*b(j,k)+c(j,i)
14            enddo
15          enddo
16        enddo
17  !$omp end parallel
18        write(*,*)c
19        end
```

Fig.4 A sample of OpenMP matrix multiplication

图 4 OpenMP 矩阵乘程序示例

该程序所使用的全部变量集合为{*a*,*b*,*c*,*i*,*j*,*k*}.如果在第 7 行进行计算状态保存,并在并行复算时从第 8 行开始执行(为简化描述,这里未考虑程序段的划分),那么由于变量 *a* 和变量 *b* 的将在第 13 行被引用,并且其值分别来自于第 5 行和第 6 行的赋值语句,而这些赋值语句在并行复算过程中不会被执行,因此 *a* 和 *b* 的值必须被保存.而变量 *i*,*j*,*k* 和 *c* 将在第 13 行被引用,其值来自于第 10 行~第 13 行的赋值,在并行复算的过程中会重新执行这几条赋值语句,所以这 4 个变量的值不需要保存.

通过上述示例,在程序中某个位置 *p* 上进行计算状态保存时,需要保存的变量 *x* 同时满足下面的条件:

- *x* 在 *p* 之后的第 1 次访问是一次引用;
- *x* 在 *p* 之前的某处被定值.

即,在 *p* 处进行计算状态保存时,需要保存的变量应该是 *p* 处的活跃变量.

程序中任一点处的活跃变量集合是程序空间中所有变量集合的一个子集,因此,保存活跃变量作为该点处的计算状态可以减少数据保存量,从而降低程序在正常执行过程中的容错开销.

2.3.2 面向 OpenMP 并行程序的活跃变量求解方法

在传统的编译技术中,使用数据流方程的迭代解来求解串行程序中任一个单入口单出口代码块 *B* 的入口处的活跃变量集合.该求解方法对于串行程序是适用的,但是对于 OpenMP 并行程序来说,由于某些变量在进入并行区后具有特殊的数据作用域属性,导致关于该变量的定值到达关系发生变化,因此活跃变量求解的结果可能不正确.例如,考虑如图 5 所示的 OpenMP 程序片段.

在图 5(a)中,*x* 在语句 10 中被定值,在语句 20 中被引用,并且从 10~20 的路径上不存在对 *x* 的定值.因此,按照先前所描述的方法,在给定的一点 *p* 处 *x* 是活跃的.但是实际上,由于 *x* 在并行区中具有 PRIVATE 属性,因此在并行区入口处,各个线程都会得到一个 *x* 的私有副本,并且其初值是未定义的.也就是说,在语句 10 中 *x* 被赋予的值不能到达语句 20,因此 *x* 在 *p* 处实际上是不活跃的.在图 5(b)中,在语句 20 引用 *x* 之前,语句 10 对 *x* 进行定值,因此在 *p* 处 *x* 是不活跃的.但同样,由于并行区中的 *x* 具有 PRIVATE 属性,所以在串行区中 *x* 在 *p* 到语句 20 之前并未被定值,因此,实际上 *x* 在 *p* 处是活跃的.

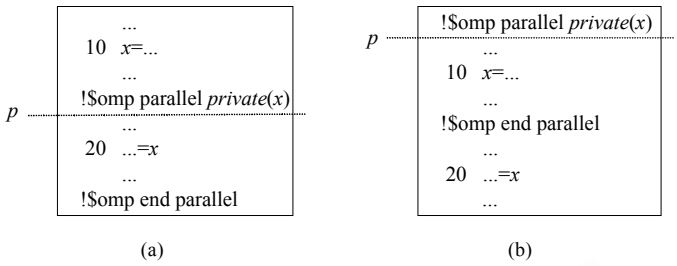


Fig.5 An OpenMP code fragment

图 5 OpenMP 程序片段

为此,本节提出一种基于 OpenMP 并行控制流图和变量数据作用域属性的数据流分析方法.首先,讨论 OpenMP 并行控制流图的概念.OpenMP 并行程序的运行遵从 fork-join 模式,其中的串行区由主线程执行,并行区由一组线程(包括主线程)并行执行.在程序运行的这段时间内,从线程执行的只是程序中被包含在并行区内的那些代码,其操作的数据集合中一部分是与其他线程共享的数据,另一部分是私有访问的数据,并且这两部分之间存在数据交互而引入新的引用定值关系,这是传统的数据流分析方法不能处理的.

为此,定义 OpenMP 并行控制流图 OMP-CFG(OpenMP control-flow graph).设 G 是一个 OMP-CFG,则 G 是一个二元组$\langle G_0,G_k\rangle$,其中,$G_0=\langle N_0,E_0\rangle$,$G_k=\langle N_k,E_k\rangle$,分别描述主线程的执行和从线程的执行(因为所有的从线程执行完全相同的代码).图中的结点对应基本块,边表示控制流中结点的先后关系.一个典型的 OpenMP 程序及其在 k 个线程上运行时的并行控制流图如图 6 所示.

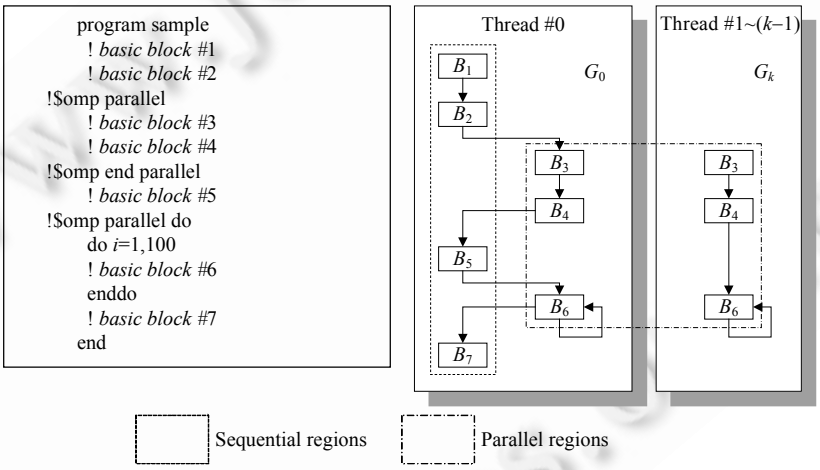


Fig.6 A sample of OMP-CFG

图 6 OMP-CFG 示例

当某个变量进入并行区时,其私有副本在生成之后会按照变量的数据作用域属性所定义的语义进行初始化.在这一过程中,该变量在串行区中共享副本的值有可能会传播到私有副本中.此外,在离开并行区时,某些数据属性会要求一个变量私有副本的值传播到其后串行区中该变量的共享副本中.这两个过程中实际上隐含着对变量共享副本和私有副本的赋值,称由此引发的定值与引用为隐式定值和隐式引用.

在图 5 所示的情形下,传统活跃变量分析结果之所以不准确,就是因为没有考虑到变量 x 数据作用域属性的语义所包含的隐式定值和隐式引用.针对这一问题,在并行控制流图上进行活跃变量分析之前,首先对流图做一次变换,根据每个并行区中变量的数据作用域属性,将其语义中隐含的赋值操作显式地添加到控制流图中该并

行区的某些基本块中,使隐式定值和隐式引用能够被包含到这些基本块的 Use 集合和 Def 集合中.

为了区分共享副本和私有副本,对于程序中任意变量 x,使用其下标形式 x_i 替换其 x 作为私有副本的所有出现,下标 i 表示 x 在并行区 P_i 中的私有副本.表 1 给出针对各种具有私有语义的数据作用域属性的赋值操作添加规则.

Table 1 Rules for adding assignments

表 1 赋值操作添加规则

x 在 P_i 中的属性	添加位置	添加操作		
THREADPRIVATE	P_i 的入口处	若 x 出现在 COPYIN 子句中		$x_i=x$
		若 x 未出现在 COPYIN 子句中	$i=1$	$x_i=0$
			$i>1$	$x_i=x$
	P_i 的出口处	$x=x_i$(仅在 G_0 中添加)		
PRIVATE	P_i 的入口处	$x_i=0$		
FIRSTPRIVATE	P_i 的入口处	$x_i=x$		
LASTPRIVATE	P_i 的出口处	$x=x_i$(仅在 G_0 中添加)		
REDUCTION	P_i 的入口处	$x_i=c$(其中,c 是由 REDUCTION 子句中所指定的运算决定的一个常数)		
	P_i 的出口处	$x=x$ op x_i(其中,op 是由 REDUCTION 子句中所指定的运算)		

按照上述方法变换后的 OpenMP 并行控制流图可以用于进行准确的活跃变量分析.分析过程分别在 G_0 和 G_k 上进行,在 G_0 上可以求解主线程任意一点的活跃变量集合,而在 G_k 中可以求解从线程上任意一点的活跃变量集合.值得注意的是,G_k 中不包含串行区中的任何基本块,所以在其上求解活跃变量集合时,也不会考虑共享变量在串行区中的引用和定值,因此,关于共享变量的分析结果是不准确的.但实际上,共享变量的活跃性在 G_0 上已经得到准确的分析,因此,应该在 G_k 上的分析得到的活跃变量集合中删除所有的共享变量.

2.3.3 优化实例

在本节中,我们以一个 OpenMP 程序片段作为实例,说明上述数据流分析方法对计算状态保存量的优化效果.该程序片段来自 NPB OMP 3.2 测试程序包[10]中的 CG 程序,如图 7 所示.其中,p_1,p_2,p_3 表示在该片段中指定的 3 个检查点保存位置.

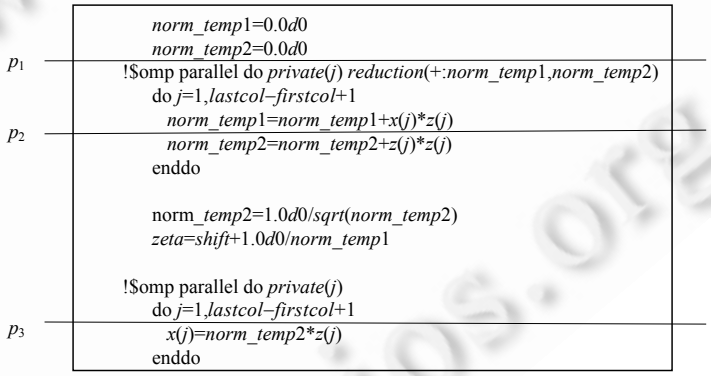


Fig.7 Main iteration of NPB OMP CG

图 7 NPB OMP CG 的主迭代

若不进行数据流分析,则在 3 个检查点位置上需要保存的变量是该程序片段中的全部变量,包括 3 个等长的数组 x,y,z 和若干标量.若一个数组的数据量为 d,则在忽略标量大小的情况下,3 个检查点的大小均为 $3d$.

按照本节所给出的数据流分析方法,在 3 个检查点位置上的活跃变量集合分别如下:

- $LiveIn[P_1]=LiveIn[B_2]=\{norm\_temp1,norm\_temp2,shift,firstcol,lastcol,x,z\}$;
- $LiveIn[P_2]=LiveIn[B_4]=\{x,z,j\_1,norm\_temp1,norm\_temp2,norm\_temp2\_1\}$;
- $LiveIn[P_3]=LiveIn[B_7]=\{norm\_temp2,j\_2,z\}$.

同样地,在忽略标量大小的情况下,3 个检查点位置上活跃变量集合所包含的数据量分别为 $2d$,$2d$ 和 d.也就是说,在这 3 个位置上进行检查点保存时,通过数据流分析可以降低约 44%的数据量.

2.4 面向fail-stop故障模型的故障检测

PR-OMP 机制作为一种快速的故障恢复方案,并不包含具体的故障报告机制,而是采用类似于中断处理的一种机制,依赖并行执行环境(如线程库和 OS)进行错误报告.PR-OMP 在每个程序段末尾调用故障检测例程,通过访问一个称为线程故障描述符(thread failure descriptor,简称 TFD)的数据结构来判断当前是否有故障发生.若参与并行执行的线程数为 k,则 TFD 为长度为 k 的向量$\langle E_1,E_2,\ldots,E_k\rangle$,其中,$E_i(1\leqslant i\leqslant k)$用于表示线程 T_i 是否发生故障.PR-OMP 面向 fail-stop 故障模型,即,在程序执行过程中一有故障发生,只要是并行执行环境能够感知到的,该故障就将被立即报告出来,并相应地更新 TFD 的值.在其后任何一个线程访问 TFD 之后,都可以确切地知道是否有故障发生,以及故障发生的位置.

故障检测需要对所有参与并行执行的线程进行一次同步.由于故障检测在程序段的末尾进行,而按照第 2.1 节提出的程序段划分方案,程序段的最后总是存在一个并行循环,因此故障检测总是在一个并行循环之后进行.如果该并行循环没有指定 NOWAIT 子句,则其出口处隐含着一次同步.因此,在调用故障例程之前,可以不需要额外再进行同步,完全没有性能损失.但如果没有 NOWAIT 子句,就必须在调用故障检测例程之前先进行同步,在这种情况下,对于负载不平衡的程序,可能引入较大的同步开销.

3 PR-OMP 的编译支持

以源到源转换的方式将一个 OpenMP 程序改造成 PR-OMP 程序,需要比较复杂的程序分析和改写,尤其是程序段的划分和状态保存时所必需的活跃变量分析.如果程序较复杂,那么这部分工作很可能会失误而影响改造后程序的正确性.为此,我们设计实现了一个源到源转换工具,称为 GiFT-OMP(get-it-FT for OpenMP),能够为 OpenMP/Fortran 77 程序加入并行复算机制.GiFT-OMP 的组成结构和工作原理如图 8 所示.

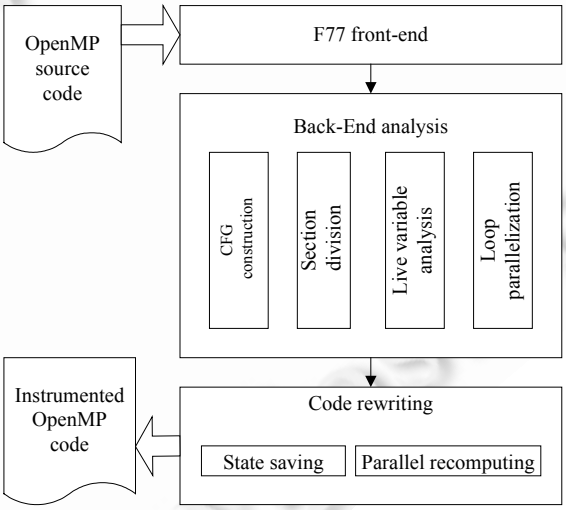


Fig.8 Structure of GiFT-OMP

图 8 GiFT-OMP 的组成结构

各个模块的主要功能如下:

- F77 前端:基于 GCC-4.3.0 gfortran 模块修改得到 F77 编译前端,前端扫描 OpenMP 源程序建立符号表、语法树以及与 OpenMP 并行机制相关的内部抽象表示;同时,对 PR-OMP 所引入的新指导语句子句

PR_FRAGS()和 *PR_SCHE*()进行解释.前端的分析结果包括一系列表格,记录诸如子程序调用关系、符号表、并行区(并行循环)位置信息、并行环境的设置与更改情况等等.

- 分析后端:在前端得到信息的基础上,后端分析模块首先建立流图,并按照第 2.1 节给出的程序段划分方法进行初始分段和修正.然后,根据第 2.3 节讨论的活跃变量分析方法求解每个程序段入口处活跃变量集合.最后,利用公式(3)求出程序中各个静态调度的并行循环中每个线程被分配的循环块,同时初始化数据结构来记录动态调度的并行循环运行时每个线程所执行的迭代集合.
- 代码重写模块:代码重写模块负责在每个程序段的起始处插入状态保存代码,将所有的活跃变量存入文件;然后在程序段的末尾插入故障检测代码;最后在每个程序段之后插入一段基于该程序段中所包含的并行循环生成的并行循环,作为并行复算代码.

如第 1.2 节所述,PR-OMP 容错机制在将程序分段的基础上将每个程序段转换成并行复算段.相应地,经 GiFT-OMP 转换后的 OpenMP 程序与原程序相比,主要新增如下几部分代码:

- 用于完成计算状态保存的代码,记为 C_s;
- 用于完成故障检测的代码,记为 C_d;
- 用于完成计算状态恢复的代码,记为 C_l;
- 用于完成并行复算的代码,记为 C_r.

这些新增的代码不会改变原有程序的正确性,其原因在于以下 3 个方面:

首先,新增的代码不会对原程序计算状态中的任何变量写入错误的值.在程序执行过程中不发生故障的情况下,C_l 和 C_r 不会被执行,而 C_s 和 C_d 不会对原程序计算状态中的任何变量进行写操作.在发生故障的情况下,C_l 对并行复算过程中所需要使用的变量进行写操作,且写入的值来自于最近一次所保存的计算状态,而该计算状态一定是正确的.因此,只要状态保存和恢复过程中 I/O 操作不发生故障,就不会对计算状态中的任何变量写入错误的值.如第 2.2.3 节所述,C_r 是以原程序中的并行循环为基础,并利用故障线程号计算出适当的迭代空间边界而生成的一段代码.因此,只要并行复算过程中不发生新的故障,则复算结果一定的是正确的.

其次,在发生故障的情况下,并行复算过程能保证所有因故障而导致结果错误的计算过程被重新执行.如第 2.2.1 节所述,并行复算的负载是由故障线程的线程号或动态生成的簿记信息所决定的.因此,在检测到故障后,发生故障的那个线程在当前所处的并行算段中被分配的所有计算任务都会被重新计算.

最后,在发生故障的情况下进行并行复算,虽然存在若干调度单元的执行顺序与原程序略有区别,但不会导致错误的计算结果.在 OpenMP 并行区中,任意两个调度单元执行的先后顺序与最终的计算结果无关,但相邻的并行区和串行区必须被顺序执行才能保证计算结果正确.而 PR-OMP 选择在程序段的末尾进行并行复算能够满足这一约束,不会因执行顺序不同而导致错误的计算结果.

4 实验评测

我们在一台 8 结点的机群系统上对 PR-OMP 的容错性能做了评测,并与 C^3 系统的容错性能进行了比较.机群系统每个结点配置 Intel® Xeon® 2.00GHz 的四核处理器和 4GB 主存,结点间使用千兆以太网互连.机群运行 Fedora 10 操作系统,内核版本 2.6.27.5,使用 gcc 4.3.0 编译器.测试程序选用的是 NPB(NASA parallel benchmark)3.2[10]测试程序包中的 FT,EP,和 IS 等 3 个程序.

根据机群的硬件条件,在实验中使用 EP 和 FT 的 B 级数据规模,IS 的 C 级数据规模,测试了两组执行时间.第 1 组是原始测试程序使用不同线程数的执行时间,第 2 组是经 GiFT-OMP 转换过的测试程序使用不同线程数在发生一次故障的情况下的执行时间.两组执行时间均是多次测试结果的平均值,并且在测试第 2 组执行时间时,在程序的每次运行时均随机选取一个程序段进行故障注入,这样得到的执行时间平均值能够比较公正地反应并行复算开销.

图 9 给出了 3 个测试程序原始版本和 PR-OMP 版本的执行时间统计,其中,深色的矩形表示原始版本的执行时间,浅色的矩形表示并行复算版本在无故障情况下的执行时间,白色的矩形表示并行复算版本在发生一次

故障后进行了并行复算的情况下的执行时间.

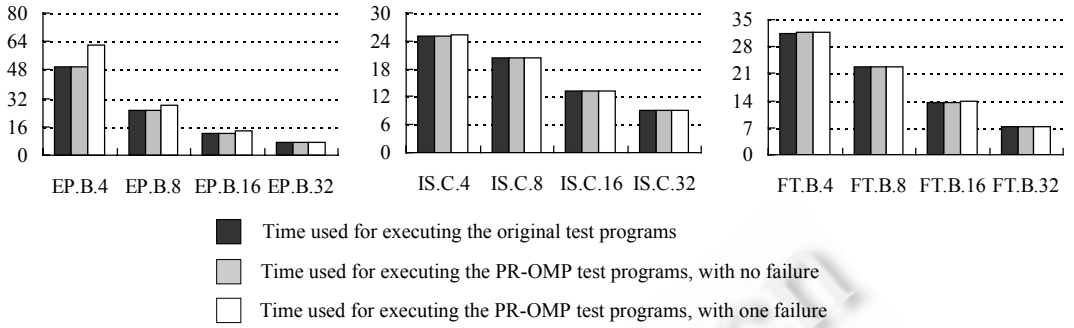


Fig.9 Execution time of test programs

图 9 测试程序执行时间统计

从图 9 可以看出,测试程序的并行复算版本在无故障情况下的执行时间与相应的原始版本的执行时间并无明显差异.这是由于经过活跃变量分析,在状态保存时需要保存的变量个数很少,对于某些并行循环来说甚至不需要保存任何变量.但是在发生一次故障的情况下,测试程序的并行复算开销则呈现出不同的情形.EP 程序相对来说并行复算开销比较大,这是由于其特殊的程序特征所致.EP 中仅包含 1 个 OpenMP 并行循环,并且该并行循环的执行时间占程序总执行时间的绝大部分.也就是说,EP 包含一个运行时间非常长的程序段,所以并行复算开销较大.如图 9 所示,EP 的容错开销在 4 个线程运行的情况下达 25%.与 EP 不同,IS 和 FT 两个程序均包含数十个并行循环.相应地,这两个程序均被划分若干为个程序段,因此容错开销维持在较低的水平.

图 10 给出了 3 个测试程序的 PR-OMP 版本相对于原始版本在使用不同线程数执行时,容错开销相对于原始版本执行时间的百分比.从中可以看出,并行复算开销随着测试程序所使用线程个数的增加有显著的降低.这是因为并行复算时所使用的线程数也相应增加的原因,这说明并行复算机制具有一定的可扩展性.

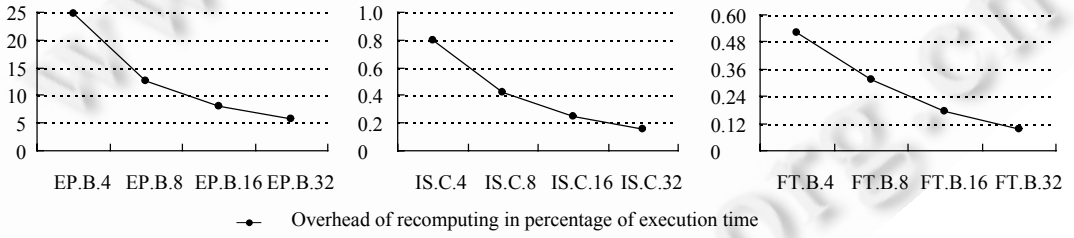


Fig.10 Overhead for recomputing varies as the number of threads grows

图 10 测试程序并行复算开销随线程数增加的变化情况

此外,实验中还测试了 3 个测试程序在使用 8 个线程执行时分别进行一次计算状态保存和计算状态恢复的开销,并与目前国际上唯一一种较为完善的应用级检查点机制——C^3 系统的检查点保存和检查点恢复开销进行对比,实验数据见表 2.其中,C^3 系统的数据来自文献[7,11].

表 2 比较了 PR-OMP 和 C^3 两种容错机制下,计算状态保存和恢复的时间占测试程序原始版本执行时间的百分比.由于这一指标与数据集规模无关,同时也由于实验环境的限制,在实验中并未刻意地使用与文献[7,11]相同的数据集规模.从比较结果可以看出,除 EP 外,PR-OMP 的计算状态保存和恢复的开销占原始版本执行时间的百分比均小于 C^3 系统的检查点保存与恢复时间占原始程序执行时间的百分比,说明本文所提出的容错机制在性能上具有一定的优势.表中给出 C^3 下 EP 的计算状态保存开销为 0.00%,而在 PR-OMP 下为 0.06%,高于 C^3 的开销.这并非与本文的结论相悖,而是由于 EP 的状态保存所需要的时间极短,而 EP 在 C 级和 B 级数据规

模下长运行时间有比较长,所以状态保存开销占程序执行时间的百分比小到可以忽略.而实验过程中任何一点时间误差都会导致百分比计算结果的偏差,因此导致了这一结果.

Table 2 Performance comparison of PR-OMP and C^3

表 2 PR-OMP 与 C^3 系统的容错性能比较

容错机制	程序	数据集	状态保存开销(%)	状态恢复开销(%)
C^3	EP	C	0.00	0.50
	FT	A	18.87	11.30
	IS	B	14.02	7.83
PR-OMP	EP	B	0.06	0.36
	FT	B	0.17	1.56
	IS	C	0.12	1.89

关于故障恢复的计算开销,由于目前 C^3 系统的相关文献中均未公布这方面的数据,因此本文目前还无法进行实际的比较.

5 结论与未来工作

基于回滚-恢复的容错技术在高性能计算领域被广泛应用.本文介绍了针对 OpenMP 并行程序的容错机制 PR-OMP,在 OpenMP 程序运行的过程中,周期性地选取必需的变量集合进行计算状态保存,如果执行过程中有线程发生故障,其他无故障线程可以从最近的状态保存点恢复计算状态,然后并行地复算故障线程上丢失的计算任务.本文讨论了并行复算机制的基本问题,包括程序段的划分、状态保存时变量集合的选定、故障线程的负载重分布以及故障检测等.本文还介绍了用于支持 PR-OMP 机制的编译工具 GiFT-OMP.该工具能够以源到源的方式将普通的 OpenMP 转换为具有 PR-OMP 机制的自容错程序.最后,通过实验对 PR-OMP 的性能进行了评测.结果表明,并行复算的开销较低,并且能够随着参与运行的线程个数增加而降低,说明 PR-OMP 具有一定的可扩展性.实验结果还显示了 PR-OMP 相对于 C^3 系统在容错性能方面的优势.

目前,状态保存时进行的活跃变量分析还不够精确,主要体现在对数组的处理上.当前,本文的分析还未能深入到数组内部,针对单个数组元素进行依赖关系和活跃性分析.如果能够解决这一问题,就能进一步降低状态保存的开销.

References:

[1] TOP500 supercomputing site. http://www.top500.org

[2] Reed DA, Lu CD, Mendes CL. Reliability challenges in large systems. Future Generation Computer Systems, 2006,22(3):293−302. [doi: 10.1016/j.future.2004.11.015]

[3] Sorin DJ, Martin MMK, Hill MD, Wood DA. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA 2002). Anchorage, 2002. 123−134. [doi: 10.1109/ISCA.2002.1003568]

[4] Prvulovic M, Zhang Z, Torrellas J. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA 2002). Anchorage, 2002. 111−122. [doi: 10.1109/ISCA.2002.1003567]

[5] Dieter WR, Lumpp JE. A user-level checkpointing library for POSIX threads programs. In: Proc. of the '99 Symp. on Fault-Tolerant Computing Systems (FTCS'99). Madison, 1999. 224−227. [doi: 10.1109/FTCS.1999.781054]

[6] Bronevetsky G, Marques D, Pingali K, Szwed P, Schulz M. Application-Level checkpointing for shared memory programs. In: Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004). New York, 2004. 235−247. [doi: 10.1145/1024393.1024421]

[7] Bronevetsky G, Pingali K, Stodghill P. Experimental evaluation of applicationlevel checkpointing for OpenMP programs. In: Proc. of the 20th Annual Int'l Conf. on Supercomputing (SC 2006). Cairns, 2006. 2−13. [doi: 10.1145/1183401.1183405]

[8] Bronevetsky G, Marques D, Pingali K, Stodghill P. C^3: A system for automating application-level checkpointing of MPI programs. In: Proc. of the 16th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC 2003). 2003.

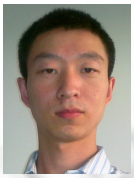
[9] Yang XJ, Du YF, Wang PF, Fu HY, Jia J, Wang ZY, Suo G. The fault tolerant parallel algorithm: The parallel recomputing based failure recovery. In: Proc. of the 16th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT 2007). Brasov, 2007. 199–212. [doi: 10.1109/PACT.2007.4336212]

[10] Bailey DH, Harris T, Saphir W, Wijngaart RVD, Woo A, Yarrow M. The NAS parallel benchmarks 2.0. Technical Report, NAS-95-020, NASA Ames Research Center, 1995.

[11] Bronevetsky G. Portable checkpointing for parallel applications [Ph.D. Thesis]. Ithaca: Cornell University, 2007.

富弘毅(1978－),男,新疆乌鲁木齐人,博士生,主要研究领域为并行与分布式系统,容错技术,科学计算.

丁滟(1977－),女,博士生,主要研究领域为可信计算.



宋伟(1981－),男,博士生,主要研究领域为并行体系结构,容错技术,科学计算.

杨学军(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为并行体系结构,并行操作系统,并行编译.