基于OpenMP/Fortran的源到源转换事务存储编程环境

黄 春 贾建斌 彭 林
(国防科学技术大学计算机学院 长沙 410073)

摘 要 首次在Fortran语言中引入事务存储,对OpenMP Fortran API进行了扩展,以源到源转换的方式实现了FortranTM编译器原型。针对软件事务存储实现的特点,扩展了EXCLUDED和SCHEDULE指导命令子句,以便为程序员提供性能调整优化API。测试结果表明FortranTM API编程便利,具有良好的性能。

关键词 源到源转换,事务存储,FortranTM
中图法分类号 TP314 **文献标识码** A

Source-to-Source Compiling Approach to Extend OpenMP/Fortran with Transactional Memory

HUANG Chun JIA Jian-bin PENG Lin
(School of Computer Science, National University of Defense Technology, Changsha 410073, China)

Abstract OpenMP Fortran directive APIs were extended to support transactional memory. This is the first time to introduce TM into Fortran language. The source-to-source translation method was involved in the compiler prototype named FortranTM. EXCLUDED clause was introduced and SCHEDULE clause was extended with transaction size parameter. Experiment results show that FortranTM interfaces facilitate transactional programming and provide impressive performance profits.

Keywords Source-to-source, Transactional memory, FortranTM

1 引言

并行编程需要更多的编程技巧和经验,编程难度也更大。自从20世纪70年代以来,人们就使用互斥锁和条件变量来控制多个线程对共享变量的访问,这种机制一直是主流的并行程序编程思想。但是基于锁的同步方式很容易导致死锁,特别是多个包含锁的程序组合在一起时,更加难以保证不发生死锁。并行程序中锁之间的关系总是难以理清,这种局面常常使并行编程面临困境。

事务存储(Transactional Memory,TM)并行编程借用了数据库中事务(transaction)这一术语及其思想。相比使用锁的同步机制,事务存储不需要程序员管理锁之间复杂的相互关系,而是依靠一种更为抽象和结构化的概念——原子操作,从而简化了并行编程。

在事务存储思想中,事务是一组必须原子执行的读写操作和运算序列。该序列要么执行完成并提交执行结果(commit),要么中止执行,丢弃已产生的结果(abort),将程序状态恢复到该序列开始执行之前,以便重新执行(retry/rollback)。只有事务提交之后,其执行结果才是可见的,任何中间状态对事务外部不可见。事务存储编程环境的实现方式有3种:基于库调用、基于程序语言扩展和基于编译器指导命令。现有的研究主要集中在C/C++和Java语言环境中,而Fortran作为广泛应用在高性能科学计算领域的一种程序语言,事务存储思想尚未引入其中。

本文描述了一种在Fortran语言中应用事务存储的方式。我们在OpenMP/Fortran指导命令的基础上扩展类似于斯坦福OpenTM[1]的事务存储指导命令,并使用源到源转换(source-to-source)的方式实现事务存储编译器原型,我们称之为FortranTM。FortranTM实现了EXCLUDED指导命令子句,并对SCHEDULE子句进行了扩展,使之在事务化循环构造中支持事务大小的设置。测试结果表明,FortranTM具有良好的性能。

2 FortranTM API和运行时接口

OpenMP[2]是共享存储多线程编程的事实标准。斯坦福大学等研究机构的研究人员提出了在OpenMP的基础上实现事务存储。本文将这一思想扩展移植到Fortran语言中。

2.1 FortranTM API

FortranTM的编程接口是OpenTM API向Fortran语言的移植,主要包括:

事务构造:声明代码块以事务化方式执行,指定了一个事务的开始和结束位置。其语法如图1(a)所示。该指导命令应该出现在OpenMP的并行区域(PARALLEL)内。

事务化循环构造:其语法如图1(b)所示。事务化循环允许程序员将循环结构放入并行区,而不用考虑循环迭代内部或迭代之间是否存在数据相关。循环迭代将作为事务执行。

到稿日期:2010-05-06 返修日期:2010-10-06 本文受国家科技重大专项(2009ZX01036-001-003),国家863高技术研究发展计划项目"面向片上多处理器系统的程序设计环境"(2008AA01Z110),国家自然科学基金(60903059)资助。

黄 春(1973-),女,博士,副研究员,主要研究方向为并行编译、编译优化技术,E-mail:chunhuang@nudt.edu.cn;贾建斌(1982-),男,硕士,主要研究方向为并行编译技术,E-mail:jj6925@gmail.com(通信作者);彭 林(1979-),男,博士,主要研究方向为编译优化技术。

事务化循环可以通过类似于 OpenMP DO 构造中相应的指导命令子句进行控制，如 SCHEDULE，ORDERED 等。

事务化 section 构造：与 OpenMP 中的 SECTION 指导命令相对应，该结构将每个 SECTION 代码分配给一个独立的线程，并且将整段 SECTION 代码作为一个事务执行。其语法如图 1(c)所示。

事务化函数声明指导命令：事务中调用的函数必须能够保证事务语义的完整性与一致性。该指导命令对这些函数进行声明，指导编译器在编译时进行处理。其语法如图 1(d)所示。

```
!$OMP TRANSACTION
      code-block
!$OMP END TRANSACTION                    (a)

!$OMP TRANSDO [CLAUSE, ...]
      do-loop
!$OMP END TRANSDO                        (b)

!$OMP TRANSSECTIONS
!$OMP TRANSSECTION
      section-block
!$OMP TRANSSECTION
      section-block
...
!$OMP END TRANSSECTIONS                  (c)

!$OMP TM_FUNCTION func_name              (d)
```

图 1　FortranTM API 语法

2.2　FortranTM 运行时接口

根据 STM 的运行机制[3]，事务化程序需要 4 组基本的运行时接口：

STM_INIT 和 STM_EXIT：初始化和退出事务执行环境。

STM_THREAD_ENTER 和 STM_THREAD_EXIT：创建和释放线程执行事务所需要的上下文环境。执行事务的所有线程都需要运行这两个接口，STM_THREAD_ENTER 在线程开始执行第一个事务之前调用，STM_THREAD_EXIT 在线程终止之前调用。

STM_BEGIN 和 STM_END：启动和提交事务，同时指明事务的边界。

STM_READ 和 STM_WRITE：对共享变量的读写操作进行包装，插桩到事务体中，保证存储一致性。

3　FortranTM 的源到源转换策略

FortranTM 在 CCRG 编译器[4]的基础上实现，其编译过程如图 2 所示。FortranTM 由两部分组成：语法分析器和代码转换器。语法分析器将 FortranTM API(包括 OpenMP 指导命令)转换为机器平台无关的二进制中间形式，输出到. dep 文件中。代码转换器重构. dep 文件，将 FortranTM API 转换为事务存储运行时库的调用，然后进行反向分析，输出标准 Fortran 程序。使用本地编译器编译输出程序，即生成事务化的执行代码。

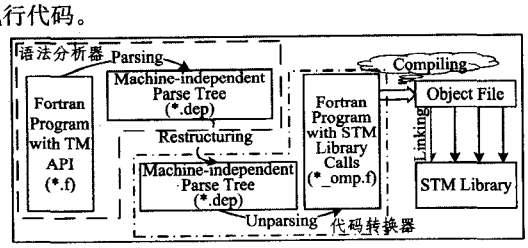

图 2　FortranTM 编译器框架

3.1　FortranTM API 转换

FortranTM 的核心是源到源转换。FortranTM API 在 OpenMP 的基础上实现，它们的转换在同一层次完成。下面描述各个接口的转换策略。为了突出 TM 转换，部分 OpenMP API 的转换在图中没有体现。

事务构造：如图 3 所示，事务入口和结束语句分别转换为运行时库接口 STM_BEGIN()和 STM_END()的调用语句。事务中的共享变量通过调用 STM_WRITE()和 STM_READ()接口进行包装，实现事务化读写插桩，如图 3 中 var1 和 var2 所示。

```
!$OMP TRANSACTION
 statements
 var1 = var2 * i
!$OMP END TRANSACTION
-----------------------------------
CALL STM_BEGIN()
 instrumented-statements
 CALL STM_WRITE( var1, STM_READ(var2)*i )
CALL STM_END()
```

图 3　事务构造指导命令的转换

事务化循环构造：图 4 展示了事务化循环的转换。循环的迭代代码被插桩为事务执行。循环任务分配和调度在 OpenMP 的基础上实现，4.2 节进行了详细讨论。

```
!$OMP TRANSDO [CLAUSE, ...]
 do-loop
!$OMP END TRANSDO
-----------------------------------
DO WHILE ( comp__type_more (comp__dolo,
        comp__dohi, comp__doin).EQ.1 )
 DO lc_i=comp__dolo, comp__dohi, 1
  CALL STM_BEGIN()
  instrumented-loop-iteration
  CALL STM_END()
 END DO
END DO
```

图 4　事务化循环指导命令的转换

事务化 section 构造：其转换策略如图 5 所示。

```
!$OMP TRANSSECTIONS                  !$OMP SECTIONS
!$OMP TRANSSECTION                   !$OMP SECTION
 section-block                        CALL STM_BEGIN()
!$OMP TRANSSECTION                    instrumented-block
 section-block                        CALL STM_END()
...                                  !$OMP SECTION
!$OMP END TRANSSECTIONS              ...
                                     !$OMP END SECTIONS
```

图 5　事务化 section 指导命令的转换

事务函数：如图 6 所示，编译器为事务函数生成两个函数体，一个经过事务化插桩，另一个未插桩，分别用于事务代码内部和外部调用。

```
!$OMP TM_FUNCTION func_name
FUNCTION func_name (var_list)
 function-body
END FUNCTION
-----------------------------------
FUNCTION func_name (var_list)
 function-body
END FUNCTION

FUNCTION TX_func_name (var_list)
 instrumented-function-body
END FUNCTION
```

图 6　事务函数的转换

线程初始化和资源释放：事务执行时线程需要维护一些元数据。而在线程结束时，这些元数据所占用的资源需要释放掉。FortranTM 在含有事务构造的并行线程启动和结束位置分别插桩运行时，库接口函数 STM_THREAD_ENTER()和 STM_THREAD_EXIT()实现线程的初始化和资源释放，它们分别作为线程所执行的第一条和最后一条语句。其插桩如图 7 所示。

```
!$OMP PARALLEL                    SUBROUTINE test__parallel_0_0(m, k, j, i)
...                                 ENTRY test__parallel_0_1(m, k, j, i)
!$OMP TRANSACTION                   CALL STM_THREAD_ENTER()
  code-block                        CALL STM_BEGIN()
!$OMP END TRANSACTION               instrumented-code-block
...                                 CALL STM_END()
!$OMP END PARALLEL                  ...
                                    CALL STM_THREAD_EXIT()
                                    RETURN
                                  END SUBROUTINE
```

图 7　事务线程初始化和资源释放例程的插桩

4　FortranTM 的性能调整 API

事务存储机制在程序中添加了大量的额外操作，导致很高的开销。例如，STM 为每个事务维护一个读写集合，访问共享变量需要对读写集合进行查找比较，从而读写集合越大，引起的开销也越多。

图 8 展示了对 STAMP[5] 的测试结果。该测试是基于 TL2 STM[6] 的。如图所示，事务化读操作（TxLoad）、事务化写操作（TxStore）以及事务提交操作（TxCommit）所占程序执行时间的比例通常达到了 60%～80%。

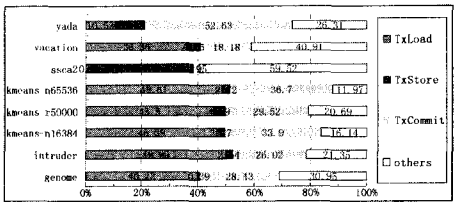

图 8　STAMP 中事务化读写与提交操作所占时间比例

降低 STM 开销，除了优化 STM 算法外，编译器还应当提供一定的支持。FortranTM 为程序员提供了以下两条 API，用于调整程序性能。

4.1　EXCLUDED 子句

事务存储系统不负责检查变量的共享或私有属性。借助于 OpenMP 辨别出来的变量属性，FortranTM 能够自动排除对私有变量的事务化读写插桩。

但是除了私有变量以外，一些共享属性的变量也不会引起数据竞争。这些变量可以分为两类：一类是只读共享变量，即在所有并行的事务中不存在对该变量的写操作；另一类是存在写操作且为单线程专有的变量（其他线程既不读也不写该变量）。如图 9 中的示例，假设图中所有变量之前被声明为共享变量。在示例(a)中，变量 varB 和 varC 都是只读的；而在示例(b)中，变量 varA 和 varX 都属于第二类单线程专有变量。

这两种类型的共享变量都不需要事务化读写插桩。FortranTM 引入 EXCLUDED[7] 子句来处理这些变量，其语法如下：

EXCLUDED(list)

list 是只读共享变量或单线程专有共享变量的列表，这些变量将被 FortranTM 的语法分析器标记。程序员必须保证列表中的变量在并行线程之间不存在任何数据竞争。图 9 也展示了 EXCLUDED 子句的用法。

```
!$OMP PARALLEL                          !$OMP PARALLEL TRANSSECTIONS
!$OMP TRANSDO EXCLUDED(varB, varC)      !$OMP+EXCLUDED(varA, varX,...)
  DO i=1,N                              !$OMP TRANSSECTION
    varA=varB+varC                        varA=varB+varC
  END DO                                !$OMP TRANSSECTION
!$OMP END TRANSDO                         varX=varX+varD
!$OMP END PARALLEL                      !$OMP END PARALLEL TRANSSECTIONS
          (a)                                     (b)
```

图 9　只读和单线程专有共享变量以及 EXCLUDED 的使用

在代码转换时，只读变量不需要任何处理。单线程专有变量需要轻量级的函数插桩，以保证事务的回滚特性。FortranTM 提供了如下两个运行时库接口：STM_WRITE_BUFFER()将这些变量的写结果加入一个缓冲区，只有事务提交时才将缓冲区写入变量的实际存储地址；STM_READ_BUFFER()从缓冲区中读取变量的值。与 STM_WRITE() 和 STM_READ()相比，这两个例程不需要对变量进行有效性验证，因而开销较小。以图 9(b)中的语句 varX＝varX＋varD 为例，该语句转换之后为：

```
CALL STM_WRITE_BUFFER( varX,
        STM_READ_BUFFER(varX)＋varD )
```

4.2　事务化循环中的事务大小

图 8 中显示了事务提交操作（TxCommit）通常占据程序执行时间的 30%。在负载一定的情况下，增大事务，程序中事务的数量相应减少，从而可以降低事务提交操作的开销。这一方法非常适合事务化循环：循环迭代的总次数是固定的，而通过改变事务中包含的迭代次数，可以灵活调整事务的大小。

OpenTM 在 SCHEDULE 子句中增加一个参数，用于定义事务化循环中的事务大小。与之类似，FortranTM 对其语法和语义进行了更严格的定义，其语法如下：

SCHEDULE(*kind*,*chunk_size*,*tx_size*)

参数 *kind* 和 *chunk_size* 与其在 OpenMP 中的含义相同。参数 *tx_size* 定义了循环中事务的大小，它的含义是事务所包含的循环迭代次数，默认值为 1。*tx_size* 的取值应该能够整除 *chunk_size*。例如，*chunk_size* 值为 100 而 *tx_size* 值为 10 时，循环中的每个事务包含 10 次迭代，线程间每个任务调度块中包含 100/10＝10 个事务。

FortranTM 采用循环展开和循环分块的方式实现事务大小语义。*tx_size* 值较小时采用循环展开，首先将循环体展开 *tx_size* 次，之后按照 3.2 节中的策略进行插桩。*tx_size* 值较大时采用循环分块，将循环的 *tx_size* 次迭代构造为一个内层循环，并把该内层循环作为一个事务。

对如下的事务化循环，采用循环分块进行事务化插桩的结果如图 10 所示。

```
! $OMP TRANSDO
! $OMP ＋SCHEDULE(kind,chunk_size,tx_size)
   do-loop
! $OMP END TRANSDO
```

```
DO WHILE ( comp__type_more (comp__dolo, comp__dohi, comp__doin).EQ.1 )
 DO lc_i=comp__dolo, comp__dohi, tx_size
  CALL STM_BEGIN()
  DO lc_i_in=lc_i, MIN(lc_i+tx_size-1, comp__dohi), 1
   instrumented-loop-iteration(s)
  END DO
  CALL STM_END()
 END DO
    END DO
```

图 10　循环分块方式的事务化循环转换

5　FortranTM 测试

本文对 FortranTM 的测试包括两方面：EXCLUDED 子句对事务化读写操作数量和程序性能的影响，以及不同事务大小对程序性能的影响。测试环境为 Intel 酷睿四核处理器，单核频率 2.6GHz，内存为 2GB DDR2 存储器，运行 Ubuntu 8.10 操作系统，本地编译器采用 GNU gfortran-4.1 版本。

5.1 EXCLUDED 子句的性能测试

我们以一个将二维矩阵乘积求和任务划分为两个事务化 SECTION 的程序测试 EXCLUDED 子句对性能的影响。在该程序中，矩阵 B 和矩阵 C 中的元素分别在两个 SECTION 中累加到共享变量 SUM 中。由于 B 和 C 在各自的线程中是只读的，因此可以使用 EXCLUDED 子句修饰。表 1 列出了使用 EXCLUDED 子句前后事务读写操作的数量，图 11 展示了该程序基于 TL2 和 TinySTM[8] 的执行时间。

表 1 EXCLUDED 子句对事务化读写操作数量的影响

是否使用 EXCLUDED	TxRead 数量	TxWrite 数量	事务化读写操作总数量
是	20000	20000	40000
否	40000	20000	60000

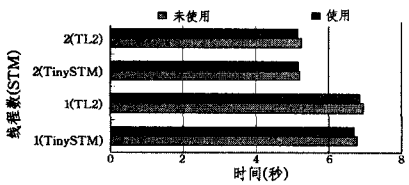

图 11 使用 EXCLUDED 子句前后的程序执行时间

从表 1 中可以看出，使用 EXCLUDED 子句有效减少了事务化读操作的数量(减少 50%)。但是由于事务化执行部分在该程序中所占比例较小，由此带来的性能提升也较小(最大 1.9%)。

5.2 事务大小对程序性能的影响

本文采用数值积分估算圆周率 pi 的程序测试不同事务大小时程序的性能表现，从中分析事务大小对程序性能的影响。该程序的主要部分是一个循环迭代：

```
h=1.0/n
pi=0.0
DO i= 1,n
  pi=pi+h*(4/(1.0+h*(i-0.5) * h*(i-0.5)))
END DO
```

程序中迭代次数 n 越大，估算出的 pi 的值精度越高。我们使用 FortranTM 事务化循环指导命令，分别将事务大小设置为 1,4,10,25,50，那么相应的每个事务包含 1,4,10,25,50 条计算 pi 的语句。由于每条语句都要读取和写入变量 pi，因此该程序执行期间事务之间的竞争非常激烈。

采用 TinySTM 运行时库和循环展开事务大小处理策略的测试结果如图 12 所示。图中显示了线程数分别为 2 和 4 时程序执行的事务统计数据，其中 aborts 表示所有线程总共产生的事务中止次数，lock-r 表示事务化读操作失败导致的事务中止，lock-w 表示事务化写操作失败导致的事务中止。

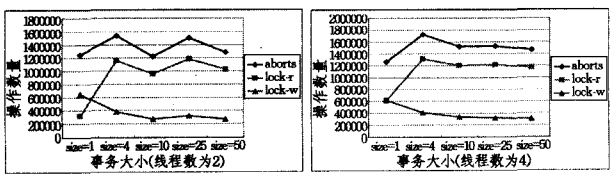

图 12 不同事务大小时 abort 的次数

abort 次数代表了程序执行时事务冲突的激烈程度。从图中可以看出，在运算任务负载不变的情况下，随着事务大小的增大，存在一个 abort 次数先变大后减小的过程。这是因为增大事务大小延长了单个事务的生命周期，使得不同线程执行的事务之间发生冲突的遭遇窗口变大，冲突的概率增大；随着事务大小增大，事务总量减少，即使事务冲突概率增大，事务 abort 的数量也比以前减小。

在上述测试条件下，程序对应的执行时间及其加速比如图 13 所示。从图中可以看出，事务大小对程序性能有显著影响，选择合适的事务大小能够获得更好的性能。与图 12 对比，当事务大小由 1 增大到 4 时，aborts 次数急剧增加，但程序仍获得了较好的加速比。这是因为 TinySTM 采用的相遇时加锁(Encounter Time Locking，ETL)策略[3] 极大地降低了事务 abort 造成的开销。

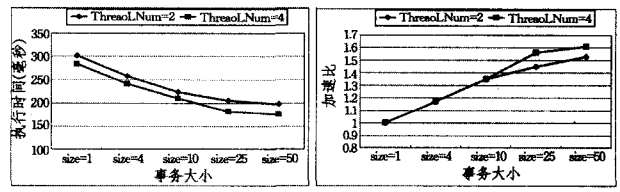

图 13 不同事务大小时的程序执行时间和加速比

结束语 本文介绍了一种设计实现事务存储并行程序编译环境的方法，并首次在 Fortran 语言中尝试了事务存储的应用。以 TL2 和 TinySTM 为事务存储运行时库，实现了 FortranTM 事务存储编程环境。测试结果表明，FortranTM 能够成功实现基本 Fortran 程序的事务化插装。FortranTM 试验了两种优化事务存储程序性能的策略，即通过 EXCLUDED 子句减少事务化读写插装，以及在事务循环中调整事务大小。这对于事务化并行程序性能调整有重要应用价值。

参 考 文 献

[1] Baek W, Minh C C, Trautmann M, et al. The OpenTM Transactional Application Programming Interface [C]// International Conference on Parallel Architectures and Compilation Techniques(PACT). Toronto, Canada, 2007: 376-387

[2] OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0 [S]. 2008

[3] Larus J R, Rajwar R. Transactional Memory(Synthesis Lectures on Computer Architecture) [M]. United States of America: Morgan & Claypool, 2007

[4] Huang Chun, Yang Xue-jun. CCRG OpenMP: Experiments and Improvements [C]// The 1st of International Workshop on OpenMP. Eugene, Oregon USA, June 2005

[5] Minh C C, Chung J W, Kozyrakis C, et al. STAMP: Stanford Transactional Applications for Multi-Processing [C]// Proceedings of The IEEE International Symposium on Workload Characterization(IISWC '08). Seattle, WA, USA, 2008: 35-46

[6] Dice D, Shalev O, Shavit N. Transactional Locking II [C]// 20th International Symposium on Distributed Computing (DISC). Springer, Stockholm, Sweden, 2006, 4167: 194-208

[7] Milovanoviç M, Ferrer R, Unsal O, et al. Transactional Memory and OpenMP [C]// Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era. Beijing, China, 2007, 4935: 37-53

[8] TinySTM [OL]. http://tinystm.org/tinystm