以状态子集为中心的并行模型检测算法①

张营飞[1,2], 谢 淼[1,2], 张 珩[1,2], 杨秋松[1]

[1](中国科学院软件研究所 基础软件国家工程研究中心, 北京 100190)
[2](中国科学院大学, 北京 100190)

摘 要: 以线性时序逻辑 LTL(Linear Temporal Logic)模型检测算法为研究对象, 提出以状态子集为中心的并行模型检测算法. 针对传统单机多核算法同步开销大的缺点, 新算法充分利用状态子集的稠密特性动态调度任务, 从而降低同步开销, 提高算法并行度. 本文基于轻量级单机图计算框架 Ligra, 结合检测过程中状态子集的特性, 设计并实现新的在线(on-the-fly)模型检测算法. 与现有算法相比, 在模型检测的效率上可以提升 20-30%, 具有高扩展性特征.

关键词: LTL 模型检测; 状态子集; 并行; 在线方法; 图计算

Parallel Model Checking Algorithm Based on State Subset

ZHANG Ying-Fei[1,2], XIE Miao[1,2], ZHANG Heng[1,2],YANG Qiu-Song[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Logic model checking raise unique challenges to the efficiency and scalability of various algorithms. In this paper, it argues that the linear temporal logic model checking algorithm calls for an efficient and common solution. We then introduces a novel parallel on-the-fly model checking algorithm based on subset of states, namely SCPLM. According to the density property of subgraph with active states subset, SCPLM schedules parallel computing tasks dynamically, which has been proven to reduce the synchronization overhead significantly. It implements SCPLM on a lightweight parallel graph processing system, Ligra. A detailed evaluation using benchmarks shows that SCPLM outperforms the existing state-of-art algorithm by up to 1.2~1.3X, yet with much better scalability.

Key words: LTL model checking; Subset of states; parallel; on-the-fly method; graph processing

1 引言

模型检测[1]为并发系统的验证提供了一种自动化地验证方法, 以逻辑表示正确性规范, 依靠有效、灵活的可达性分析进行自动化验证, 目前已广泛应用于软硬件的可信性验证. 但是模型检测本身面临状态空间组合爆炸问题(State Space Explosion)[2]. 缓解模型检测状态爆炸问题的方法, 可以分成三种类型, 串行优化方法, 分布式方法, 多核并行空检测方法. 本文的研究问题是如何设计一种高效、可高扩展性的并行在线(On-the-fly)显示状态(Explicit)线性时序逻辑(LTL)模型检测算法, 在一定程度上通过多核计算平台来缓解组合爆炸问题.

首先, 在 LTL 模型检测领域, 串行优化算法主要分为两类[5] NDFS(Nested DFS)方法和基于 SCC (Strongly Connected Components)的方法, 这种方法一般能在线性时间内完成问题验证, 但是由于没有利用到多核并行, 处理状态爆炸问题有一定局限性. 其次, 也有很多研究设计出分布式模型检测算法, 能够充分利用多台计算机内存和计算资源将大规模的模型检测问题压缩到几个小时或者是几天之内处理完, 但是分布式模型检测方法机器之间的通信开销很大. 最后, 考虑到最近硬件性能越来越强, 很多分布式模型检测

① 基金项目:国家自然科学基金(91318301);中国科学院战略性科技先导专项(XDA06010600)
收稿时间:2016-01-29;收到修改稿时间:2016-03-03 [doi:10.15888/j.cnki.csa.005361]

算法可以基于多核并行来提高处理速度，在共享内存系统中，这些算法由于很低的通信开销而变得更快.

对于单机多核并行 LTL 模型检测方法. Homlmann 和 Bosnacki[3]提出基于 SPIN 的单机多核模型检测算法，而算法由于是基于深度优先算法(DFS)，难以并行化，从而扩展性很差，因此，J. Barnat 基于广度优先算法(BFS)将 OWCTY[4]在单机上并行上实现并取得不错的效果[8]，减少线程之间的竞争，通常整个状态空间需要切分给各个线程，这种切分在一定程度上减少了竞争但引入的切分开销很大.

针对上述问题，本文基于单机图计算框架 Ligra[12]，设计并实现了以状态子集为中心的并行在线 LTL 模型检测算法，我们将其称为 SCPLM(Subset of States Centric Parallel LTL Model Checking). SCPLM 分为两部分：状态空间生成和可接收环检测. 我们在线生成状态空间，满足一定条件发起可接收环检测. 在可接收环检测时，我们发起广度优先遍历(BFS)，根据遍历过程中产生的状态子集(BFS 每一层遍历后活跃的点集)的稠密度来选择按边遍历还是按顶点遍历(详细请见后面第三部分)，这样能实现动态的任务调度，降低了同步开销，提高了并行度. 因为 LTL 模型检测可以规约为在状态空间中对可接收环检测问题，一个可接收环即为反例，所以当检测到可接收环时退出程序，否则继续生成状态空间后继续检测.

我们以顶点子集为中心的并行模型检测算法有如下优点:

① 高性能，根据遍历过程中产生的状态子集的稠密度来动态的选择遍历方式，从而降低同步开销，提升效率.

② 平衡性，不用设计复杂的 hash 函数提前将状态分配给每个线程，对每轮遍历产生的状态子集里的状态动态的分配给线程做处理，分配更加平衡，因此提高了并行度，进而提升了效率.

③ 基于轻量级单机图计算框架 Ligra，使得所提出的算法易于实现，屏蔽了并行计算底层细节.

2 相关工作

Clarke et al. 提出 LTL 模型检测能够规约成一个环检测问题，并使用串行算法嵌套深度优先[1](Nested DFS)来完成检测，然而 Nested DFS 需要维护 DFS 序列，这样便让此种串行算法难以并行. Barnat et al. 提出第一个分布式模型检测算法[6]，基本思路是利用一个全局数据结构保存 DFS 后序序列，但是此方法在很多例子仍然只能串行计算. Brim et al. 又提出在带权有向图中的负环检测[7]，在生成带权有向图之后，只需要用一个单源最短路径算法就可以找到可接受环. Černá 和 Pelánek 实现分布式 OWCTY[4]算法，在很多场景表现出很高的的扩展性. 然而，这两种算法都不支持在线验证，所以只能在生成全部状态空间之后才能发起可接受环检测. Brim et al.提出基于反向边的分布式模型检测[9],它在生成状态的同时发起 BFS 遍历来检测可接受环. 这些分布式算法通信开销都很大.

Homlmann[3]提出基于 SPIN 的单机多核模型检测算法，而算法由于是基于深度优先算法(DFS)，难以并行化，从而扩展性很差，因此，J. Barnat 基于广度优先算法(BFS)将 OWCTY[4]在单机上并行上实现并取得不错的效果[8]，减少线程之间的竞争，但是整个状态空间需要切分给各个线程，这种切分在一定程度上减少了竞争但引入的切分开销很大.

本文我们提出以状态子集为中心的在线并行模型检测算法，SCPLM 有如下优点. 首先，在线的模型检测算法往往只需生成小部分状态空间，就可检测到模型不满足性质. 其次，在可接受环检测时，我们发起 BFS，根据遍历过程中的状态子集的稠密度来选择遍历方式，很大程度上降低了同步开销，提高检测效率. 再次，我们对状态子集中的状态动态均衡分配给线程并行处理，提高了并行度. 最后，基于轻量级单机图计算框架 Ligra，屏蔽了并行计算底层细节，使得所提出的算法易于实现.

3 Ligra图处理框架

Ligra[12]是单机多核环境下基于共享内存的轻量级图数据处理框架，在处理图的广度优先遍历(BFS)问题是时能够根据遍历过程中产生的顶点子集的稠密性来动态的调整遍历方式，减小同步开销，提高并行度，从而保证遍历的高效性.

Ligra 提供如下数据结构和接口:

① Vertex subset，表示计算过程中产生的顶点子集. 为了保证动态给线程分配顶点，在计算过程中要求每轮处理一个顶点子集.

② VERTEXMAP，通过一个用户自定义函数来处理顶点子集里的顶点.

③ EDGEMAP, 根据计算过程中的顶点子集 V 的稠密度来选择按边还是按顶点来遍历 V 的后继节点. 此处稠密度的度量方法是 V 的所有出边累加和, 大于阈值(threshold = M/α, 其中 M 为整个图的边个数, α 为给定常数)为稠密, 小于阈值为稀疏. V 为稠密时, 如果按边来遍历 V 所连的顶点集将产生很多竞争, 需要的同步开销大, 如果按照顶点遍历则没有竞争的产生. 另外 V 的稠密度为稀疏时, 按边遍历产生竞争的概率很小, 因此选择按边遍历. 所以这种动态的根据顶点子集的稠密性遍历可以减少竞争, 降低同步开销. 另外在使用此接口时, 需要实现 UPDATE 和 COND 接口, UPDATE 用来更新遍历到的顶点的状态(遍历过, 未遍历过), COND 用来加快遍历, 直接已遍历过的顶点.

简而言之, Ligra 根据计算过程中产生的顶点子集的稠密度来选择按边遍历还是按点遍历, 这种方法能够减少遍历过程中的竞争次数, 减小同步开销, 从而提高遍历效率. 每轮只处理一个 Vertex subset, 这样保证了动态的给线程分配顶点, 从而保证了分配的均衡性, 提高了并行度.

算法 1 展示了一个利用 Ligra API 完成的广度优先搜索示例的伪代码.

本文将研究如何设计一个以状态子集为中心的 LTL 模型检测算法, 能够充分发挥 Ligra 的计算优势, 缓解模型检测的组合爆炸问题.

算法 1 BFS()

输入:图 G, 根节点 r, 输出: 无

```
1.  Parents ← {-1,…,-1}
2.  bool UPDATE(s,d)
3.    return (CAS(&Parents[d],-1 ,s ))
4.  bool COND(i)
5.  BFS(G, r)
6.    Parents[r] ← r
7.    Frontier ← {r}
8.    while (SIZE(Frontier) ≠ 0) do
9.      Frontier ← EDGEMAP(G, Frontier, UPDATE, COND)
10.   end while
```

4 Ligra中的LTL模型检测

LTL 模型检测方法将模型检测问题规约成一个 Büchi 自动机的空检测问题, 即软件系统模型和 LTL 否定式 Büchi 自动机的同步积 Büchi 自动机, 而空性检测是搜索积空间是否有可接收环.

4.1 LTL 模型检测相关定义

定义 1(标记迁移系统(LTS)). 一个标记迁移系统可以表述成四元组(S, s_0, L, δ),其中 S 是有限状态集, s_0 ∈S 是初始状态, L 是转移标记的集合, δ 表示状态间的迁移关系, δ: S×L×S. 我们用 $s\xrightarrow{\alpha} s'$来表示,当 α∈L 时 s 转移到状态 s'.

定义 2(LTS 并行组合(LTS1 ‖ LTS2)). P1= (S_1,s_{01},L_1,δ_1), P2= (S_2,s_{02},L_2,δ_2), P1 和 P2 并行组合后的 LTS =(S_1×S_2,(s_{01},s_{02}),L_1∪L_2,δ), 其中 δ 的定义如下:

$$\frac{s_1\xrightarrow{\alpha}s'_1 \wedge s_2\xrightarrow{\alpha}s'_2}{(s_1,s_2)\xrightarrow{\alpha}(s'_1,s'_2)},\quad \alpha\in L_1\cap L_2$$

$$\frac{s_1\xrightarrow{\alpha}s'_1}{(s_1,s_2)\xrightarrow{\alpha}(s'_1,s_2)},\quad \alpha\in L_1\backslash L_2$$

$$\frac{s_2\xrightarrow{\alpha}s'_2}{(s_1,s_2)\xrightarrow{\alpha}(s_1,s'_2)},\quad \alpha\in L_2\backslash L_1$$

定义 3(Büchi 自动机(BA)). BA 可以用一个五元组(Q,Σ,δ,q_0,A)表示, 其中 Q 是有限状态集, Σ 是字母表, q_0∈Q 是初始状态, A⊆Q 是可接收状态集合, δ 表示转移关系, δ： Q×Σ×Q. BA 是有限自动机能够接收无限的输入, 当且仅当一个无限输入序列中存在至少一个接收状态时, 这个无限的输入序列是可接收的.

M(M=M_1‖M_2…‖M_n)是一个系统, F 是一个 LTL 公式. 模型检测问题是看是否 M⊨F, 表示 M 的所有行为都满足 F 的可接收状态. 要处理这种问题, 根据自动机理论首先用¬F 产生一个 BA[15], M⊨F 取决于 M⊗¬F=∅, ⊗表示 LTS 和 BA 的同步积, 下面给出同步积的定义.

定义 4(LTS 与 BA 的同步积(LTS⊗BA)). LTS= (S_1,s_{01},L_1,δ_1), BA=(Q,Σ,δ_2,q_0,A), 同步积也是一个自动机, 用 LTS⊗BA= (S×Q,Σ',δ',I',A)表示, 其中 Σ'=Σ∩L, I'={(s_0,q_0)|∃α(α∈Σ∧$s_0\xrightarrow{\alpha} s'$∧$q_0\xrightarrow{\alpha} q'$)}此时 s'∈S,q'∈Q, δ'定义如下:

$$\frac{s\xrightarrow{\alpha}s' \wedge q\xrightarrow{\alpha}q'}{(s,q)\xrightarrow{\alpha}(s',q')},\quad \alpha\in\Sigma'.$$

当且仅当系统中包含一个非法行为时同步积自动机为空, 为了验证是否有非法行为, 需要在对应的状态图中检查是否存在一个包含可接收状态的环, 如果存在, 表明有非法行为, 反之没有.

4.2 SCPLM 算法框架

SCPLM 的输入是一个由标记迁移系统刻画的模

型 M 和一个 LTL 公式 F, 目的是验证 M⊨F 是否成立, 如果不成立, 输出反例. 为了进行 LTL 模型检测, SCPLM 会检查自动机的同步积 M⊗¬F 是否至少存在一个可接收环. 因此算法设计的关键点在于可接收环检测.

SCPLM 主要分成两部分, 状态空间生成和可接收环检测. 在可接收环检测时, 我们发起 BFS 遍历, 根据遍历过程中产生的顶点子集的稠密度(如第三章所述)来动态的选择按边遍历还是按顶点遍历, 从而保证并行高效的可接收环检测. 根据顶点子集的稠密度来选择遍历方式是利用 Ligra 的 EDGEMAP 接口完成的, 对于具体的可接受环检测, EDGEMAP 中需要自定义遍历过程中 UPDATE 函数.

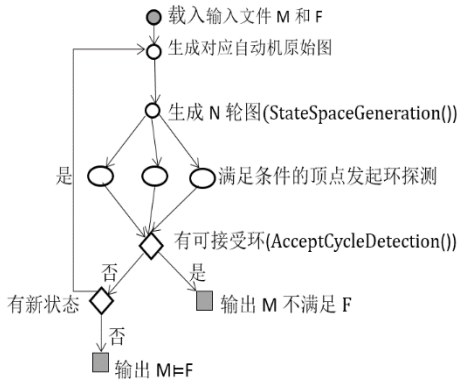

图 1 SCPLM 框架图

SCPLM 的执行流程如图 1,首先输入模型文件 M 和 LTL 公式 F, 根据 M 和 F 生成对应的自动机原始图和起始节点. 然后, SCPLM 调用函数 StateSpaceGeneration()以在线(on-the-fly)的方式迭代的生成 N(N 为给定常数)轮状态空间, 对满足一定条件的顶点(详细见下一节)在已生成的状态空间构成的图里发起可接收环检测操作, 可接收环检测由函数 AcceptCycleDetection()发起. 如果存在可接收环, 输出反例并结束程序, 否则, 检查是否还有新的状态产生, 没有就结束程序, 有就继续循环的进行下一个 N 轮图的生成和可接收环检测. 如果程序检测到可接收环或者状态空间的状态全部生成, 那么程序就结束. 其中两个函数的详细设计在下一节给出.

5 SCPLM算法

这部分将重点介绍我们算法的状态空间生成和探测可接收环探测部分的设计细节.

5.1 基本概念

前面我们讲过满足一定条件发起可接收环检测, 这个条件为当生成的状态空间构成的图中出现反向边时, 反向边为存在可接受环的必要条件, 具体见如下定义和分析.

定义 5(层数). 给定一个状态转移图 G=(S,E)以及一个初始状态 s_0 和任意状态$\{s_u|s_u\in S\wedge s_u\neq s_0\}$. s_u 的层数就是 s_0 到 s_u 的最短路径, 用 $d(s_u)$来表示, 其中 $d(s_0) = 0$.

定义 6(反向边). 给定任意一个在图 G 中的状态 s_u 和它的直接后继节点 s_v, 一条边$(s_u,s_v)\in E$ 是一个反向边当且仅当 $d(s_u)\geq d(s_v)$. s_u 是反向边的初始状态, s_v 是目的状态.

基于上述定义我们将通过定理 1 说明反向边是图中环存在的一个必要条件.

定理 1. 给定一个状态转移图 G=(S,E)以及一个初始状态 s_0, 对于在图中的每一个环 $\Omega=s_m,\ldots,s_n$, $n>m\geq 0, m\geq 0, |\Omega|\geq 1$, 存在一个 j, $0\leq j\leq n$ 满足 $d(s_j)\geq d(s_i)$, 其中 $0\leq i\leq n$. 在环中从 s_j 发出的任何一条边都是反向边, 层数 $d(s_j)$为这个环的最大深度.

一个环 Ω 是可接收环当且仅当在这个环里有一个可接收状态. 因此从定理 1 来看, 一个可接收环肯定至少包含一个反向边和一个可接收状态. 所以可接收环的一个必要条件为反向边.

5.2 状态空间的生成算法

算法 2 表示的状态空间生成的过程, 起始我们设置了一个可调节的迭代轮次常数 N, 表示每次生成 N 轮的状态图空间, 之后再在反向边的起始状态发起可接收环检测, 并行地生成状态空间, 并行部分(第 5 行)调用并行库 CILKPLUS[11]来完成. 我们用 BFS 生成状态空间, 作为起始状态集 initQueue 开始只有一个初始状态, 随着状态的生成 initQueue 里存的是每轮迭代生成最新的后继状态. 算法的输出为生成的全局状态图 G, 在后续的可接收环探测就是在 G 中发起的, 输出还有作为下 N 轮状态图生成的起始顶点集 nextQueue, 以及发起的可接收环探测反向边初始点集 blsQueue.

如果 initQueue 起始状态集不为空并且此 N 轮图未生成结束就继续生成(1-3 行), 然后并行的对每一个 initQueue 顶点调用 getSuccessors 函数[10]生成后继状态. 将生成的后继点加入到 G 中, 如果点是新生成的, 将其加入到 nextQueue 用于生成下一轮图(6-9 行), 否则将其加入到反向边起始状态集 blsQueue 中. 最后将

生成的新状态存入 initQueue 继续迭代生成新状态.

反向边作为可接收环的必要条件, 需要对每个反向边的起点发起检测可接收环操作. 当然, 如果 G 中不存在可接收状态, 那么也就不存在可接收环.

算法 2 StateSpaceGeneration()

输入:M 和 F 对应的初始图 G_M 和 G_F, 起始状态集 initQueue, 以及需要 N 轮迭代

输出: 同步积对应的全局状态图 G, 下个 N 轮图的起始状态集 nextQueue,反向边起始状态集 blsQueue

```
1.  i = 0
2.  blsQueue←NULL
3.  while initQueue.isNotEmpty() and i< N do
4.      i←i+1
5.      parallel_for all v ∈ initQueue do
6.          Vsucc←getSuccessors(v)
7.          G.AddVertex(Vsucc)
8.          if  Vsucc in G then
9.              blsQueue.push(v)
10.         else
11.             nextQueue.push(Vsucc)
12.         end if
13.     end parallel_for
14.     initQueue←nextQueue
15. end while
```

5.3 基于状态子集的可接收环检测算法

blsQueue 中保存的是反向边的初始点集, 对于每个点 v∈blsQueue 我们都会调用一次可接受环检测算法. 具体方法是, 对于点 v∈blsQueue, 我们会从 v 发起一次 BFS 遍历, 如果能再次遍历到点 v, 那么表示存在一条环.

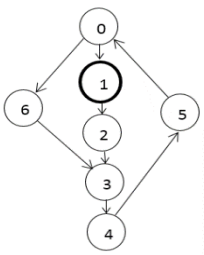

图 2 可接收环检测

为了确定在已生成的图中是否有可接收环的存在, 一个简单的方法是当发现环时, 再去遍历此环看是否存在可接收状态, 如果有证明找到了可接收环. 然而, 把查可接收状态放到查环的过程中将更为高效.

为了在查环的过程中查可接收状态, 需要为每个顶点设置一个访问状态标识 Visit, 它表示的是从初始顶点到当前顶点是否存在路径, 以及此路径中是否存在可接受状态.

下面给出 Visit 的定义. G=(V,E,s,Acc)为一个自动机图, Acc 为可接收状态集合, s 为遍历的起始点, 在查找可接收环时访问的每个点 v 有一个访问状态标志 Visit[i], 如果通过(u,v) ∈ E 边遍历到 v, 那么 Visit[v] 的定义如下:

$$\text{Visit}[v]=\begin{cases}1,\ v\in \text{Acc}, \text{或者 Visit}[u]=1\\ 0,\ v\text{被访问过}, v\notin \text{Acc}, \text{Visit}[u]\neq 1\\ -1,\ v\text{ 未被为访问过}\end{cases}$$

即如果顶点 v 没有被访问过, 访问标志为-1; 如果被访问过但是祖先顶点和自己中没有可接收状态, 访问标志为 0; 如果自己或者祖先顶点中有可接收状态, 访问标志为 1. 简而言之, Visit[v]为 1 时表示 s 到 v 的路径中存在可接收状态, 为 0 时表示路径中不存在可接收状态, 为-1 时表示还没有访问到 v. 因此在查环的过程中, 从反向边的起始点出发遍历再次找到自己, 并且在到达子集的路径上存在可接收状态, 则找到可接受环.

如图 2 查可接收环的过程, (4,5)为反向边, 从 4 发起可接收环检测操作, 知两条路径 P1(4-5-0-6-3-4)与 P2(4-5-0-1-2-3-4)中有环存在, P1 中没有可接收状态, 而 P2 中顶点 1 为可接收状态, 遍历过程中顶点 2,3,4 的访问标志会随着遍历过程更新为 1, 因此当再次遍历到 4 时, 会知道所经历的路径中存在可接收状态, 从而找到可接收环.

算法 3 AcceptCycleDetection ()

输入:全局状态图 G, 反向边的初始点集blsQueue

输出: 是否存在可接收环

```
1.  isAcceptCycle ← false
2.  AcceptCycleDetection (G,blsQueue)
3.    for T∈blsQueue do
4.      Visit← {-1,…,-1}
5.       Visit[T] ← isAccept(T)
6.       Frontier ← {T}
7.       while (SIZE(Frontier) ≠ 0) do
```

```
8.          Frontier ← EDGEMAP(G, Frontier,
    UPDATE, T, COND)
9.          if isAcceptCycle==true then
10.           report “存在可接收环” and exit
11.         end if
12.       end while
13.   end for
```

```
1.  bool UPDATE(s,d)
2.    if(Visit[d]== -1) then
3.      if(state[d] == ‘A’ || Visit [s] == 1) then
4.        Visit[d] ← 1
5.      else
6.        Visit[d] ← 0
7.      end if
8.      return 1
9.    else
10.     if((T == d) && (Visit [s] == 1 || Visit [d] == 1))
    then
11.       isAcceptCycle ← true
12.     else if(Visit [s] == 1&& Visit [d] != 1) then
13.       Visit [d] ← 1
14.       return 1
15.     end if
16.     return 0
17.   end if
```

```
1.  bool COND(d)
2.    return (!isAcceptCycle)
```

可接收环检测算法伪代码展示在算法 3 中，分为三部分，主函数 AcceptCycleDetection，以及 EDGEMAP 要调用到的子函数 UPDATE 和 COND.

AcceptCycleDetection 的输入为到目前为止已经生成的状态空间图 G 和反向边起始点集合 blsQueue，我们对每个反向边都发起一次查环操作(23 行). 对于单次查环操作，G 中每个顶点对应的访问标记都为-1(24 行)，对于反向边起始点 T 发起遍历进行可接受环检测，为了保证后面遍历时 UPDATE 操作的一致性，我们提前设置 T 对应访问标志 Visit[T]，如果为可接收状态则置为 1，否则置为 0(25 行). 像算法 1 一样，将 T 放到 Frontier 之后开始遍历操作(26-28 行)，不同的是 EDGEMAP 的遍历过程中的更新函数 UPDATE(2-19 行)与检查函数 COND(20-21 行).

其中函数 UPDATE 的功能是，在遍历过程根据前文 Visit[v]的定义将访问标志 Visit 更新. 对于一条边(s,d),当 s 在 Frontier 中时，如果 d 未被访问过，将 d 加入下一轮的 Frontier 中，当顶点 d 的状态为可接收状态或者它的父亲的 Visit[s]为 1(表示在从 T 到 s 的路径中有可接收状态)时，将 d 对应的状态标识 visit[d]置 1，两个条件都不满足时将 visit[d]置 0(3-9 行). 如果 d 已经被访问过，并且又遍历到初始点 T，并且遍历回来的路径上有可接收状态存在，表示找到可接收环(11-13 行). 如果 d 已经被访问过,并且没有发现可接收环而到 T 到 s 的路径上存在可接收状态，若 Visit[d]不为 1 将 Visit[d]置 1(13-16 行)，同时将 d 再次加入到下一轮的 Frontier 中，图 2 的点 3 就满足这种情况.

函数 COND 表示查到可接收环便可以停止任何点加入到 Frontier 中，然后结束程序.

我们对反向边起始点集的发起的可接受环检测操作是串行的(22 行)，即查完一个初始点再查另一个. 由于我们单次查可接收环操作时，EDGEMAP 根据 Frontier 稠密度动态的调整遍历方式，单次查环高度并行，同步开销小. 如果像 VCPLM[10]的基于反向边的核心算法一样，对 N 条反向边并行的发起 N 个查环操作，同步开销很大，查环效率低.

5.4 复杂度分析

假设 m 是边的个数, n 是顶点的个数. 算法 3 的时间复杂度是 O(m*(m+n))，空间复杂度是 O(m+n). 状态空间生成的算法时间复杂度跟生成的顶点和边的个数成线性关系为 O(m+n). 每条反向边最坏情况下需要查整个图，复杂度为 O(m+n)，最坏情况下右 m 条反向边，因此查环复杂度为 O(m*(m+n)). 对于空间复杂度，所有的顶点和边都需要保存在内存中，因此空间法复杂度为 O(m+n).

6 实验与分析

我们主要关注性能和可扩展性两个性能，测试了处理不同的 LTL 模型检测的运行时间及不同线程个数的运行时间.

实验平台为 Ubuntu 14.04 64-bit 操作系统，内存为 8G，CPU 型号为 Intel(R) Core(TM) i7-2600，主频为 3.4GHz，有 8 个逻辑 CPU. 实验数据为模型检测的基准数据 BEEM[13]，主要测试的模型为哲学家就餐问题，测试了处理不同个数的哲学家所需要的时间. 对于模

型我们主要验证两种 LTL 属性, 活性(liveness)满足和不满足两种情况. 表 1 展示的是所有的模型和属性. 我们基于 Ligra 实现 SCPLM.

表 1 模型和 LTL 公式

LTL 属性	属性编号	是否满足
GF(!someone.eats)	0	是
GF(philosopher[0].eats)	1	否

为了得到内存占用情况, 我们记录了不同参数、模型和属性下可达的状态数和迁移数(表 2). 在查可接收环时有一些额外的信息, 如反向边的个数, 可以看做是发起了多少次查可接收环操作. 为了说明 SCPLM 的性能优势, 我们对比了单机版 Divine 核心算法 OWCTY[4], 使用静态切分的方法来切分状态空间. 同时, 为了说明 SCPLM 中串行对反向起始点集边发起可接受环检测的贡献, 我们参考 VCPLM[10]基于 Ligra 实现并行的对所有反向边发起查可接收环操作的算法 VCPLM+.

表 2 SCPLM 的实验数据

P 个数	属性编号	bl 数	迁移数	迭代轮	可达状态数
15	0	0	9207771	8	3326720
15	1	6423	3041989	3	544749
12	1	3012	466261	3	95281

表 3 运行时间对比

属性编号	P 个数及时间(s)			算法
	15	13	12	
0	86	40	12	Divine
0	64	25	8	VCPLM+
0	60	24	7.6	SCPLM
1	55	24	5	Divine
1	47	16.4	7.1	VCPLM+
1	35	9.8	4.9	SCPLM

我们测试了 15、13、12 个哲学家在两种 LTL 公式下用所用的时间, 这里我们每生成 N 轮图之后统一发起查环操作, N 取 3. 为了衡量可扩展性, 我们测量了 15 个哲学家在在不同线程个数下所运行的时间, 然后求出加速比(图 3、4、5、6) .

SCPLM 如表中所示, 在解决 10^{12} 规模的模型检测问题时可以用很短的时间来解决. 我们知道如果模型满足给定的 LTL 属性, SCPLM 将生成所有状态, 对于 15 个 P 和属性 0(15,0)如表 2 所示没有产生反向边, 因此算法相当于是一个状态生成器. 如图 3 和 4 所示, 由于没有反向边的产生 VCPLM+和 SCPLM 都没有发起可接收环检测操作, 因此运行时间和加速比基本相同. 与 Divine 对比可以发现, SCPLM 有 20-30%的性能提升, 出现这种情况的原因是进行算法优化, 底层采用并行库 CILKPLUS 动态的分配状态, 从提高并行度, 提升效率和可扩展性.

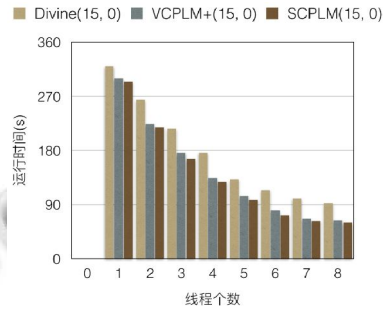

图 3 (15,0)的运行时间对比

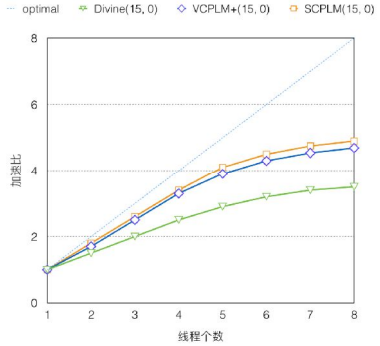

图 4 (15,0)的加速比对比

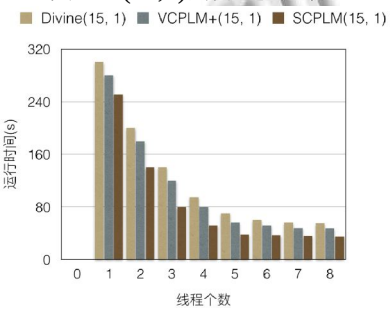

图 5 (15,1)运行时间对比

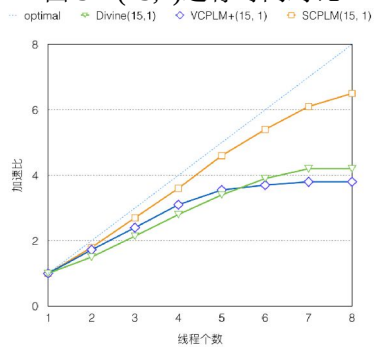

图 6 (15,1)加速比对比

由于在线算法模型不满足给定的 LTL 属性时, 其不需要产生所有状态便可以查到环. 例如(15,1)如图 5 和 4 所示, 对比性能, SCPLM 相比 Divine 快 20s(30-40%), 说明基于状态子集的查环算法利用状态子集的稠密性动态计算, 减少了竞争, 同步开销小, 性能高. VCPLM+比 Divine 的运行时间快 8s(10-20%), 进一步印证基于状态子集的查环算法的高效, 但是加速比相比于 Divine 先好, 后差, 原因是随着线程的增多同步开销进一步增大, 从而导致可扩展性差. 而 SCPLM 的性能明显好于 VCPLM+, 说明对单步检测可接受环操作比并行检测多个接收环操作更为高效, 提升(20-30%), SCPLM 的可接受环的方式竞争更少, 同步开销小, 可扩展性和性能都比 VCPLM+好. 对于扩展性, 如图, SCPLM 好于 VCPLM+和 Divine. SCPLM 在性能和可扩展性都有明显的优势, 原因是利用状态子集的稠密稀疏特性进行动态的计算, 降低了同步开销, 并且每轮动态均衡的的分配状态, 提高了并行度, 因此性能更好和可扩展性更高.

7 结论

本文提出以状态子集为中心的并行在线 LTL 模型检测算法 SCPLM, 在检测可接收环时, 据遍历过程中产生的状态子集的稠密度来选择遍历方式, 这样能实现动态的任务调度, 降低了同步开销, 提高了并行度, 从而保证高效的模型检测验证. 借助 Ligra, 我们能够高效的解决模型检测问题. 实验表明, 这种基于状态子集的稠密度来遍历的模型检测算法性能和可扩展性非常好. SCPLM 的可接收环检测是在反向边的基础上发起的, 通过实验发现单步反向边查环由于减少竞争而非常高效. 综上所述, SCPLM 探索了以状态子集为中心的 LTL 模型检测算法, 为 LTL 模型检测问题的高效解决提出新的思路.

参考文献

1 Clarke EM, Grumberg O, Peled DA. Model Checking. Cambridge: MIT Press, 1999: 13–139.

2 Katoen JP. Principles of Model Checking. The MIT Press,, 2008.

3 Holzmann GJ, Bošnački D. Multi-core model checking with SPIN. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE, 2007. 1–8.

4 Černá I, Pelánek R. Distributed explicit fair cycle detection (set based approach). Lecture Notes in Computer Science, 2003, 16(8): 0–4.

5 Zhao L, Zhang J, Yang J. Advances in on-the-fly emptiness checking algorithms for Büchi automata. 2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI). IEEE. 2012. 113–118.

6 Barnat J, Brim L, St J, et al. Distributed LTL model-checking in SPIN. Model Checking Software. Springer Berlin Heidelberg, 2001: 200−216.

7 Brim L, Černá I, Krčál P, et al. Distributed LTL model checking based on negative cycle detection. Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001). Springer Berlin Heidelberg. 2001. 96–107.

8 Barnat J, Brim L, Ročkai P. Scalable shared memory LTL model checking. International Journal on Software Tools for Technology Transfer, 2010, 12(2): 139–153.

9 Barnat J, Brim L, Chaloupka J. Parallel breadth-first search LTL model checking. Proc. of 18th IEEE International Conference on Automated Software Engineering. 2003. 106–115.

10 Xie M, Yang Q, Zhai J, et al. A vertex centric parallel algorithm for linear temporal logic model checking in Pregel. Journal of Parallel & Distributed Computing, 2014, 74(11): 3161–3174.

11 Leiserson CE. The Cilk++ concurrency platform. Journal of Supercomputing, 2010, 51(3): 244–257.

12 Shun J, Blelloch GE. Ligra: A lightweight graph processing framework for shared memory. ACM Sigplan Notices, 2013, 48(8): 135–146.

13 Pelánek R. BEEM: Benchmarks for explicit model checkers. Proc. of the 14th International SPIN Conference on Model Checking Software. Springer-Verlag. 2007. 263–267.