논리 프로그램의 AND 병렬 수행을 위한 정형적인 제안

장 병모, 최 광무

한국과학기술원 전산학과

A Formal Scheme for AND Parallel Execution of Logic Programs

Chang, Byeong-Mo, Choe Kwang-Moo

Department of Computer Science, Korea Advanced Institute of Science and Technology

요 약

본 논문에서는 논리 프로그램의 AND 병렬 수행을위한 하부구조 (framework)를 논리적인 정의를 통해서 제안한고 제안된 하부구조를 기초로하여 지능적인 resetting 방법을 제안한다 여기서 제안된 지능적인 resetting 방법은 실패한 리터럴의 바인딩 정보를 기초로하며, 이 정보는 resetting 할 때 지능의 주된 요인이다. 우리들은 제안된 지능적인 resetting 방법의 정확성을 framework에 기초하여 간단하고 논리적인 방법으로 증명한다

I INTRODUCTION

잘 알려진 것처럼 논리 언어의 단점 중의 하나는 느린 수행 속도이다 이러한 단점을 해결하기 위한 방안으로서 많은 병렬 수행 방법이 제안되었다. [CDD85, CON83, DeG84, KAL87, L&K86] 그들 방법의 대부분은 논리 프로그램에 내재된 두 종류의 병렬성, 즉 AND-병렬성과 OR-병렬성을 이용한다. AND-병렬성은 한 clause내의 두 개 또는 그 이상의 subgoal들이 병렬 수행 가능한 경우에 발생한다[C&K85] OR-병렬성은 하나의 goal을 위한 여러개의 clause을 병렬적으로 수행하여 goal의 해답들을 병렬적으로 구할 수 있게한다[CON83, CRA85]

논리 프로그램의 병렬 수행에 관한 대부분의 논문에서는 그들의 방법이 알고리즘적인 방법으로 표현되어있다 그러나 알고리즘적인 방법으로는 그들의 방법을 이해하기 힘들고 서로 비교하기 힘든 문제점을 가지고있다 이러한 문제점은 아직 논리 프로그래의 병렬 수행을 위한 정형적인 framework이 존재하지 않기 때문이다.

본 논문에서는 정형적인 정의을 통해서 AND 병렬 수행을 위한 정형화된 framework을 제안한다 또한 제안된 framework을 기초로 하여 지능적인 resetting을 위한 방법을 제안한다

제안된 지능적인 resetting 방법은 실패한 literal의 binding 정보을 이용한다. binding 정보는 지능적인 resetting의 주된 요인이다 과거에 실패한 literal의 binding 정보는 reset literal에서 실패한 해답이 다시 사용 가능한 지를 결정하는데 사용된다 제안된 지능적인 resetting 방법의 정확성은 간단하고 정형적인 방법으로 증명된다

II PRELIMINARIES

C는 n개의 body literal을 갖는 하나의 clause를 나타낸다 L_i 는 clause내의 하나의 literal을 나타낸다 특히 L_0는 clause C의 head literal을 나타낸다 L_i에 있는 변수들은 두 종류, 즉 생산 변수(generating variable)와 소비 변수(consuming variable)로 분류될 수 있다 어떤 literal L_i에 있는 변수는 binding 값이 literal L_i에 의해서 생산되면 생산 변수라 부르고, 다른 literal에 의해서 생산된 binding 값을 literal L_i가 소비하면 소비 변수라고 부른다 만약 clause body내의 두 개 이상의 literal이 변수 x_k를 공유하면 그 중 하나는 x_k의 binding 값을 생산하고 다른 literal들은 그 값을 소비한다 이 생산자 소비자 관계는 데이타 의존성 분석(data dependency analysis)에 의해서 결정된다[CDD85, C&K85] L_i의 생산 변수, *gen_variable*s은 다음과 같이

정의된다

Definition 2.1 gen_variables(L_i)
gen_variables(L_i) = {x_k| x_k *is a variable in* L_i, *and a binding value of* x_k *is generated by* L_i}

비슷한 방법으로 L_i의 소비 변수, *con_variables*은 다음과 같이 정의된다

Definition 2.2 con_variables(L_i)
con_variables(L_i) = {x_k| x_k *is a variable in* L_i, *and* L_i *consumes a binding value of* x_k *generated by other literal*}

위의 정의를 기초로하여 generator와 consumer관계를 나타내는 data dependency graph(DDG)는 directed acyclic graph로 정의된다
Definition 2.3 DDG_C
L_i와 L_j는 *clause C*내의 *literal*이라고 가정한다 *graph*의 *edge*는 *direction*을 갖는다고 가정한다
DDG_C = (V,E) *where*
V = {L_i | L_i *is a literal in a clause C*}
E = {$(L_i L_j)$ | *gen_variables*(L_i) *con_variables*(L_j) ≠ ∅ }

두 literal, L_i와 L_j 사이에 공유된 변수들은 DDG_c의 $(L_i L_j)$에 대해서 정의될 수 있다 *shared_variables*$((L_i L_j))$가 literal L_i와 literal L_j 사이에 공유 변수들을 나타낸다고 하자.
shared_variables($(L_i L_j)$) = *gen_variables*(L_i) ∩ *con_variables*(L_j)
*DDG*의 acyclic 성질은 정상적인 data dependency analysis에 의해서 보장된다[CDD85, C&K85,X&G88] 한 clause C내의 literal들은 data dependency partial order에 의해서 선형적으로 순서가 정해지고 만약 $i < j$이면 L_i는 L_j를 linearly oredered literals list(*LOLL*)에서 앞서있다

III. A FRAMEWORK FOR AND-PARALLEL EXECUTION

어떤 literal의 해답은 그 literal의 변수에 대한 *substitution*이고 앞에서 정의된 변수의 각 class에 대해서 정의될 수 있다 *generating solution* $\sigma_i^{current}$는 *gen_variables*(L_i)내의 변수들에 대한 *substitution*이고 σ_i^j는 literal L_i의 j번째로 생성된 generating solution이라고 하자.

Definition 3.1 Generating Solution $\sigma_i^{current}$
$\sigma_i^{current}$ *is a substitution* θ = {t_k/x_k | $x_k \in$ *gen_variables*(L_i), *and* t_k *is its ground binding value or* λ *when a value is not bound yet* }

어떤 clause C의 현재의 해답 $\sigma_C^{current}$은 C 내의 literal들의 generating solution들의 합집합으로 정의된다.

Definition 3.2 $\sigma_C^{current}$
어떤 *clause C*는 n개의 *body literal*을 갖는다고 가정하자
When a literal L_f *fails,*
$\sigma_C^{current} = \sigma_1^{current} \cdot \sigma_{f-1}^{current}$
Otherwise, $\sigma_C^{current} = \sigma_1^{current} \sigma_n^{current}$

L_i의 generator literal들이 *con_variable*내의 변수에 대해서 binding 값을 생산할 때마다, literal L_i는 그들에 대한 binding 값들을 받는데 이를 L_i의 consuming solution β_i라고 부른다

Definition 3.3 Consuming Solution β_i *of* L_i
β_i *is a substitution* θ = { t_k/v_k | $v_k \in$ *con_variables*(L_i), *and* $t_k/v_k \in \sigma_C^{current}$ }

$\sigma_C^{current}$는 clause C의 global environment로 생각될 수 있고 어떤 literal L_i의 consuming solution β_i은 L_i의 local environment로 생각될 수 있다 따라서 어떤 literal L_i가 어떤 local environment β_i에서 실패했다면 똑 같은 상황 β_i이 다시 주어진다면 L_i는 당연히 다시 실패할 것이다

어떤 literal의 실패한 해답은 실패할 때마다 저장될 필요가 있는데 이러한 generating solution들의 list를 L_i의 *Solutions*라고 정의한다

Definition 3.4 Solutions of L_i
L_i가 m개의 *solution*을 생산했다고 가정한다
Solutions(L_i) = (σ_i^1, σ_i^m)

제안된 방법에서는 backtracking이 일어날 때마다 *binding set* Θ라 불리우는 정보를 유지한다 Θ는 redo를 야기시킨 literal들과 그들이 소비하는 변수의 binding으로 이루어져 있다 각각의 소비되는 변수의 binding은 각 literal이 실패할 때마다 형성된다 소비된 변수의 binding중에서 backtrack literal $L_{backtrack}$의 오른쪽의 generator을 갖는 것들은 binding set Θ에 포함될 필요가 없다 우리는 이러한 두 집합의 pair을 L_f의 *Failure Information Pair*라고 부른다

Definition 3.5 Failure Information Pair(FIP)
L_f가 실패하고 어떤 *literal* $L_{backtrack}$으로 *backtrack*한다고 가정한다
FIP(L_f) = <L, Θ>
where L *is a set of redo caused literals of* $L_{backtrack}$, *and* Θ *is a set of the bindings for the variables in con_variables*(L_i) *whose generators are to the left of the redone literal* $L_{backtrack}$ *in LOLL for all* $L_i \in L$

backtracking이 일어날 때마다 $FIP(L_f)$는 backtrack literal의 현재의 generating solution $\sigma_{backtrack}^{current}$의 *Failure Cause Set*에 더해진다

Definition 3.6 Failure Cause Set (FCS)
L_r의 현재 *solution*이 $\sigma_r^{current}$일 때, L_f가 실패하고 L_r로 *backtrck*한다고 가정한다 초기값은 $FCS(\sigma_r) = \emptyset$
$FCS(\sigma_r^{current}) = FCS(\sigma_r^{current}) \cup \{ FIP(L_f) \}$

FCS내의 FIP는 만약 FIP의 모든 redo caused literal들이 현재의 해답 $\sigma_C^{current}$에서 다시 실패한다면 valid하다고 한다 이러한 관점에서 보면 FIP는 그것의 Θ가 현재의 해답 $\sigma_C^{current}$의 부분 집합이면 valid하다

Definition 3.7 Valid Failure Cause Set(VFCS)
clause C의 현재의 *solution*은 $\sigma_C^{current} = \sigma_1^{current} \cdots \sigma_{i-1}^{current}$이라고 가정한다
$VFCS(\sigma_i^j) = \{ <L, \Theta> \mid <L, \Theta> \in FCS(\sigma_i^j), \Theta \subseteq \sigma_C^{current} \}$

IV INTELLIGENT RESETTING

이 장에서는 resetting 과정을 설명하고 binding 정보에 기초한 resetting 방법을 설명한다 먼저 resetting 과정을 고려해 보자 어떤 literal L_f가 실패할 때 어떤 backtrack literal L_r이 선택되고 redo된다 그 literal L_r이 redo된 후에 $L_{r+1}, \cdots, L_{i-1}$들은 차례로 reset된다 여기서는 $L_{r+1}, \cdots, L_{i-1}$이 reset 되었고 이번에는 L_i가 reset될 차례라고 가정하자. 이제는 L_r이 redo 되었고 $L_{r+1}, \cdots, L_{i-1}$는 reset 되었기 때문에 $\sigma_C^{current}$는 $\sigma_1^{current} \cdots \sigma_{i-1}^{current}$로 변하였다.

우리들은 L_i내의 모든 해답을 검토하고 그 해답이 재사용 가능한 지 검토한다 L_i의 해답 σ_i^j을 검토할 때 $FCS(\sigma_i^j)$내에 $\sigma_C^{current}$ 하에서 valid한 FIP 존재하는지 검토한다 만약 $FCS(\sigma_i^j)$에 valid한 것이 존재하면 σ_i^j은 재사용될 필요가 없다 그렇지않으면 σ_i^j는 새로운 환경에서 성공할 지 실패할 지를 구별할 수 없으므로 재사용되어야한다

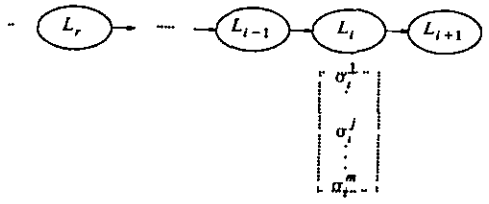

Figure 3.1 Resetting Procedure

여기서는 resetting 과정을 아래의 정리 *Resetting*으로 간단히 정리해 본다

Theorem 4.1 *Resetting*
L_r이 *redo* 되었고 $L_{r+1}, \cdots, L_{i-1}$이 *reset*되었다고 가정한다 따라서 $\sigma_C^{current} = \sigma_1^{current} \cdots \sigma_{i-1}^{current}$로 변하였다
σ_i^j *need not be re-available if* $VFCS(\sigma_i^j) \neq \emptyset$

Proof L_r이 *redo* 되었고 $L_{r+1}, \cdots, L_{i-1}$이 *reset*되었다 따라서 $\sigma_C^{current} = \sigma_1^{current} \cdots \sigma_{i-1}^{current}$로 변하였다 만약 변한 환경 $\sigma_C^{current}$ 하에서 $\sigma_i^{j'}$가 L_i의 해답으로 다시 주어진다면 변한 환경 $\sigma_C^{current}$에서 $VFCS(\sigma_i^j)$의 모든 *redo caused literals*은 다시 실패하고 L_i로 *backtrack*한다

따라서 $VFCS(\sigma_i^j) \neq \emptyset$일때 , σ_i^j을 다시 이용해도 $\sigma_C^{current}$ 하에서 실패할 것이다 결국 σ_i^j는 $VFCS(\sigma_i^j) \neq \emptyset$ 이면 재사용될 필요가 없다

V. CONCLUSION

AND 병렬 수행을 위한 많은 연구들은 *AND* 병렬 수행을 알고리즘적인 방법으로 다루었기 때문에 여러 문제점을 갖고 있다 이 논문에서는 *AND* 병렬 수행의 분명한 이해를 위해서 *AND* 병렬 수행을 위한 정형적인 *framework*이 제안되었다 제안된 *framework*은 *AND* 병렬 수행을 위한 여러 연구들의 비교 분석에 사용될 수 있다

또한 제안된 *framework*을 기초로하여 지능적인 *resetting*을 위한 방법을 제안하였다 제안된 지능적인 *resetting* 방법은 실패한 *literal*의 *binding* 정보를 최대한 이용한다 실패한 *literal*의 *binding* 정보는 제안된 방법의 지능성의 주된 요인이 된다 제안된 방법의 정확성은 *framework*에 기초하여 간단하고 정형적인 방법으로 증명된다

REFERENCES

[C&D85] Chang, J H, and Despain, A., "Semi-intelligent Backtracking of Prolog Based on Static Data Dependency Analysis", *Proc of the 1985 Symp on Logic Programming*, pp 10-21, July 1985

[CDD85] Chang, J H, Despain, A, and DeGroot, D, "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis", *Compcon Spring 85*, pp. 218-225, 1985

[CON83] Conery, J, "The AND/OR Process Model for Parallel Interpretation of Logic Programs", *Ph D dissertation, Technical Report 204, U C Irvine*, June 1983

[C&K85] Conery, J, and Kibler, D, "AND Parallelism and Nondeterminism in Logic Programs", *New Generation Computiong, Vol 3*, pp 43-70, Springer-Verlag, 1985

[CRA85] CRAMMOND, J, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages", *IEEE Trans on Computers*, October 1985.

[DeG84] DeGroot, D, "Restricted And-Parallelism", *ICOT International Conference on Fifth Generation Computer Systems*, pp 471-478, 1984

[KAL87] Kale, L V, "The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs," *Proc. of the Fourth International Conference on Logic Programming*, The MIT Press, pp. 616-632, Melbourne, Australia, May 1987

[L&K86] Lin, Y J, and Kumar, V, "An Execution Model for Exploiting AND-Parallelism in Logic Programs", *AI TR86-34, Artificial Intelligence Laboratory, The Univ of Texas at Austin*, Sept, 1986, Appendix D

[WIN87] Winsborough, W, "Semantically Transparent Selective Reset for AND Parallel Interpreters Based on the Origin of Failures", *Proc. of 1987 Symposium on Logic Programs*, pp 134-152, September 1987