Dynamic Parallel Evaluation of Cross Product Set Using Timestamps

Gopal Gupta†

Abstract

In this paper we consider the problem of dynamically generating the cross-product (cartesian product) of two or more sets, in parallel, on shared memory machines. Dynamic generation means that the cross-product set is computed on the fly as various elements of its constituent sets are generated asynchronously. Dynamic evaluation of the cross-product set finds application in a number of problems such as parallel evaluation of data-base joins. Generation of the cross-product set using the straight-forward *nested-loop* algorithms is inherently sequential because when elements of two constituents sets are generated simultaneously computing the tuples of cross-product set corresponding to these elements has to be done sequentially. We present a simple modification of the nested loop algorithm using time-stamps which results in significant amount of parallelism being extracted. We present results of experiments done on the Sequent Symmetry multiprocessor which show that the overhead introduced for time-stamping is minimal while speed-ups obtained are linear.

Keywords: Databases, Parallel algorithms.

1. Introduction

Given k sets $S_1, S_2, \ldots, S_k$, of cardinality $n_1, n_2, \ldots, n_k$ respectively, the cross-product set of $S_1, S_2, \ldots, S_k$ is defined as a set cp of ordered tuples where

$$cp(S_1, S_2, \ldots, S_k) = S_1 \times S_2 \times \ldots \times S_k = \{\langle x_1, x_2, \ldots, x_n \rangle \mid x_1 \in S_1,\ x_2 \in S_2,\ \ldots,\ x_k \in S_k\}$$

The cardinality of $cp(S_1, S_2, \ldots, S_k)$ is given by $\Pi_{i=1}^{k} n_k$.

The need to compute the cross-product of sets arises in diverse applications such as relational database query processing [U88] and parallel execution of logic programs [GJ89, RK89]. Given that the cardinality of the cross-product set is an exponential function of cardinality of its constituent sets, its computation is a time-consuming operation. Thus, it is only natural to apply parallel processing techniques for its evaluation. There are many applications in which the elements of the sets $S_1, S_2, \ldots, S_k$ are asynchronously generated over a period of time by different processes (we call such sets *dynamic* sets). In such a situation we would like the tuples of the cross-product of these sets to be available as soon as the constituent elements of the tuples have been generated. The scenario that we envisage here is that the elements of the constituent sets are asynchronously generated by multiple processes running in parallel and the tuples of their cross-product set are concurrently generated, and

† Author's Address: Dept. of Computer Science, New Mexico State University, Las Cruces, NM 88001.

perhaps further processed, by other processes also running in parallel. This scenario captures a general paradigm of parallel computing where a problem is decomposed into several sub-problems each of which have a number of solutions which can be pursued in parallel. The final solutions are obtained by combining the individual sets of solutions of the sub-problems through a cross-product operation and further processing of the combinations (cross-product tuples) thus obtained. To ensure maximum use of resources in a multiprocessor system we would like to pipeline the generation of the tuples of the cross product set (and their further processing) with the generation of the solutions. This parallel pipelined generation of the cross-product set is termed its dynamic generation.

In this paper we present an efficient technique that uses time-stamps for dynamic computation of cross-product of dynamically generated sets. Since the constituent sets are assumed to be dynamically generated in an asynchronous fashion, the technique is also applicable to dynamically generated infinite sets. Our experiments show that the technique yields linear speed-ups and incurs minimal overhead.

Throughout this paper we assume that the sets $S_1, S_2, \ldots, S_k$ are stored in a central place accessible to all the processes (or processors) in the shared memory architecture.

2. Parallel Dynamic Generation of Cross-product Set

The simplest and most obvious way of computing the cross-product of k sets would be to use a loop with k levels of nesting as follows:

```
for each x_1i ∈ S_1
   for each x_2i ∈ S_2
      for each x_3i ∈ S_3
            . . .
            for each x_ki ∈ S_k
               output(⟨x_1i, x_2i, x_3i ..., x_ki⟩);
```

Program 1: Nested Loop Algorithm for Cross-product generation

The above algorithm requires the sets $S_1, S_2, \ldots, S_k$ to be completely defined, i.e. it can be invoked only after generation of all the k sets is over, and thus is unsuitable in a dynamic scenario. In the above form it is also inherently sequential. We modify the above algorithm to yield an algorithm which does not require the sets to be completely defined. The modified algorithm is also amenable to parallel execution.

Define *partial set* $S_l(t)$ of a dynamic set S_l at time t as:

$S_l(t) = \{e \mid e \in S_l$ and e is generated at time t$'$ where $t' \leq t\}$

Note that $S_l = S_l(t = \infty)$, i.e. the final value of a dynamic set S_l is equal to the value of its partial set at $t = \infty$. The partial set $S_l(t)$ is also referred to as the *partial value* of the set S_l at time t.

Define *Partial Tuple Set* of an element e_{ji}, generated at time t and belonging to partial set $S_j(t)$, as the set of tuples

$$PTS(e_i, t) = \{\langle x_{1i}, x_{2i}, \ldots, x_{(j-1)i}, e_{ji}, x_{(j+1)i}, \ldots, x_{ki}\rangle \mid x_{li} \in S_l(t)\ (l \neq j)\}$$

The PTS of an element is empty if any of the partial sets is empty. In other words, a partial tuple set of an element is the set of all tuples which contain the newly generated element, $e_{ji} \in S_j(t)$, and which can be formed at the time that element was generated. That is,

$$PTS(e_{ji}, t) = S_1(t) \times S_2(t) \times \ldots \times S_{j-1}(t) \times \{e_{ji}\} \times S_{j+1}(t) \times \ldots \times S_k(t)$$

Note that the PTS of an element depends upon the time t, i.e. the time at which the element was generated. For example, given three sets, S_1, S_2 and S_3 with partial values $S_1(t) = \{a_1, a_2\}$, $S_2(t) = \{b_1, b_2\}$ and $S_3(t) = \{c_1, c_2\}$ where the element $c_2 \in S_3$ is generated at time t then

$$PTS(c_2, t) = \{\langle a_1, b_1, c_2\rangle, \langle a_1, b_2, c_2\rangle, \langle a_2, b_1, c_2\rangle, \langle a_2, b_2, c_2\rangle\}.$$

If, however, S_2 had accumulated more elements before time t (say $S_2(t) = \{b_1, b_2, b_3\}$) then

$$PTS(c_2, t) = \{\langle a_1, b_1, c_2\rangle, \langle a_1, b_2, c_2\rangle, \langle a_1, b_3, c_2\rangle, \langle a_2, b_1, c_2\rangle, \langle a_2, b_2, c_2\rangle, \langle a_2, b_3, c_2\rangle\}$$

The cross-product set can now be defined as:

$$cp(S_1, S_2, \ldots, S_k) = \cup_{\forall e} PTS(e, cr(e))$$

where $cr(e)$ is the time at which e was generated. The set union is performed over PTS of all elements, e, of all sets.

Theorem: The disjoint union of Partial Tuple Sets of all elements yields the correct cross-product set.

Proof: We omit the proof since it is quite straightforward. It is given in [G90].

The cross-product set can be computed in parallel by computing the various PTSs in parallel. As soon as an element of a set is generated by a process, its PTS is computed (possibly by another process) and added to the existing cross-product set. PTSs of different elements are computed by different processes giving rise to parallelism. The nested loop algorithm presented above can be trivially modified to compute the PTSs of an element $e_{ji} \in S_j$ generated at time t.

```
for each x_{1i} ∈ S_1(t)
        . . .
      for each x_{(j-1)i} ∈ S_{(j-1)i}(t)
         for each x_{(j+1)i} ∈ S_{(j+1)i}(t)
                . . .
              for each x_{ki} ∈ S_k(t)
                 output(⟨x_{1i}, ..., x_{(j-1)i}, e_{ji}, x_{(j+1)i}, ..., x_{ki}⟩);
```

Program 2: Algorithm for computing a PTS

Note that the depth of nesting is only $k - 1$ (`for` loop for set S_j is omitted) and every tuple output has the element e_{ji} in its jth position. The code given above, however, seems unrealistic since (i) the number of nested for loops depends on the number of sets and (ii) for

each element the `for` loop to be omitted is different. However, it is easy to design a general algorithm which overcomes these difficulties and which can be used for generating the PTSs of all elements (such an algorithm is given in [G90]).

Although the above technique seems simple there is still one major problem to tackle—the computation of the union of various PTSs. Computing the union, which involves checking for duplicated tuples, is an expensive task.

Duplicate tuples can easily arise if two or more elements of a set are generated at the *same time*. For example, consider two partial sets $S_1(t=2) = \{\mathtt{a}_1, \mathtt{a}_2\}$ and $S_2(t=2) = \{\mathtt{b}_1\}$. Suppose the elements $\mathtt{a}_3 \in S_1$ and element $\mathtt{b}_2 \in S_2$ are simultaneously generated at $t=3$ by two different processes. Thus, now $S_1(t=3) = \{\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3\}$ and $S_2(t=3) = \{\mathtt{b}_1, \mathtt{b}_2\}$. Clearly, $PTS(\mathtt{a}_3, t=3) = \{\langle \mathtt{a}_3,\ \mathtt{b}_1\rangle,\ \langle \mathtt{a}_3,\ \mathtt{b}_2\rangle\}$, and $PTS(\mathtt{b}_2, t=3) = \{\langle \mathtt{a}_3,\ \mathtt{b}_2\rangle,\ \langle \mathtt{a}_2,\ \mathtt{b}_2\rangle,\ \langle \mathtt{a}_1,\ \mathtt{b}_2\rangle\}$. Note that the tuple $\langle \mathtt{a}_3,\ \mathtt{b}_2\rangle$ is common to both tuple-sets. In general, if more elements are generated simultaneously, more tuples would be duplicated and many tuples may be replicated more than twice.

Duplicate tuples can also arise if elements are added to a set while PTS of an element is being computed. For example, in the above scenario if b_2 was added to S_2 while the PTS of a_3 was being computed, then the tuple $\langle \mathtt{a}_3,\ \mathtt{b}_2\rangle$ may be duplicated in this case also. Thus, even though two elements e_i and e_j may not be generated at precisely the same time elements in the cross-product set could still be replicated if e_j was generated while e_i was computing its PTS, and e_j started computing its PTS concurrently. Our experiments showed that upto 20-30% of elements in the cross-product set can be replicated for medium sized sets.

Checking for duplicates can be avoided if we can ensure that they do not arise. In such a case all PTSs would be disjoint, and the set-union will amount to a simple constant-time juxtaposition operation.

In order that duplicated tuples are avoided, all the sets should be frozen while the PTS of an element is being computed. Sets can be frozen, for instance, by obtaining exclusive access to them via a lock while an element is being added and its PTS being computed. However, this would completely sequentialize the generation of cross-product set and put us back to square one. In the next section we propose a better way to do this using time-stamps.

3. Computing Cross-Product Set with Time-stamps

Both the problems mentioned above can be avoided if we associate *unique* time-stamps with each set element. Since the generation of PTS of an element e at time t requires the value of the sets at that time, we can stamp all elements with the time at which they were generated to identify the elements which were in existence before time t. Thus time stamps let the process computing $PTS(e,t)$ identify the set-elements which are relevant for its computation without resorting to freezing the state of all sets. The relevant elements are precisely those elements which have a time stamp t_s where $t_s < t$. The uniqueness property of time stamps eliminates the duplicates generated due to simultaneously produced elements.

The algorithm for generating $PTS(e,t)$ is modified so that it only considers those elements of the sets whose time stamp is less than t. Time is maintained as a global shared counter variable whose value is read and tagged to the element. The time counter is then incremented. The reading of the time-counter and its incrementing has to be done atomically, i.e. exclusive access via a lock has to be obtained while carrying out the time-stamping operation. This results in some loss of parallelism, which is insignificant compared to that in the solution mentioned at the end of previous section†.

In the method just described the locking of large shared data-structures (sets in this case) is traded for time-stamping. The technique of trading semaphore locks for time-stamps is not new and has been used earlier in the context of concurrent databases [BG80].

The algorithm used for computing the PTS of an element $e_h \in S_j$ is shown in Program 3. Assume that `time` is the global shared variable used for generating time stamps; function `assoc(`e_h`, time)` associates time-stamp `t` with element e_h and adds e_h to the appropriate set; `tstamp(`x_{ki}`)` returns the time stamp associated with element x_{ki}; and `lk` is a semaphore lock.

```
P(lk);
assoc(e_h , time);
time := time + 1;
V(lk);
for each x_1i ∈ S_1 and tstamp(e_h) > tstamp(x_1i)
   for each x_2i ∈ S_2 and tstamp(e_h) > tstamp(x_2i)
          . . .
         for each x_(j-1)i ∈ S_(j-1) and tstamp(e_h) > tstamp(x_(j-1)i)
            for each x_(j+1)i ∈ S_(j+1) and tstamp(e_h) > tstamp(x_(j+1)i)
                   . . .
                  for each x_ki ∈ S_k and tstamp(e_h) > tstamp(x_ki)
                     output(⟨x_1i, ..., x_(j-1)i, e_h, x_(j+1)i, ..., x_ki⟩);
```

Program 3: Computing a PTS with Time-Stamps

Notice that in the above algorithm the actual cross product computation, which forms the bulk of the work, is done outside the exclusive access portion of the code, and thus can be executed by different processors at the same time. Reading and incrementing the time-stamp variable is the only operation which needs to be done sequentially in this algorithm. This sequential part can be executed in constant time and thus entails only a minimal overhead.

3.1. An Improvement

The algorithm presented above can be further improved. We can avoid obtaining exclusive access to the shared global counter used for time-stamping during generation of

† Exclusive access to an individual set may also be required when an element is added to the data-structure used for representing sets. However, this depends on the data-structure used for representing sets and is not essential since multi-port data structures [J89] may be used.

$PTS(e_h, t)$, where $e_h \in S_j$, by imposing an arbitrary total order among the sets $S_1, S_2, \ldots, S_k$ and replacing the $>$ operator by $\geq$ operator in the `for` loops corresponding to sets ranked higher than S_j in the program segment above. The algorithm used for generating $PTS(e_h, t)$ is shown below (Program 4). We assume that `time` is maintained as a clock which keeps incrementing automatically without the intervention of processes reading it (such a self-incrementing clock can be implemented as a hardware counter or by a dedicated process). The ordering among sets is assumed to be $S_i < S_j$ if $i < j$.

```
assoc(e_h , time);
for each x_1i ∈ S_1 and tstamp(e_h) > tstamp(x_1i)
   for each x_2i ∈ S_2 and tstamp(e_h) > tstamp(x_2i)
         ......
            for each x_(j-1)i ∈ S_j-1 and tstamp(e_h) > tstamp(x_(j-1)i)
               for each x_(j+1)i ∈ S_j+1 and tstamp(e_h) ≥ tstamp(x_(j+1)i)
                     ......
                     for each x_ki ∈ S_k and tstamp(e_h) ≥ tstamp(x_ki)
                        output(⟨x_1i, e_h, ..., x_ki⟩);
```

Program 4: Computing a PTS without Locking the Shared Variable

In absence of unique time stamps a tuple gets replicated if it contains more than one element generated at the same instant (as discussed in section 2). Such replicates would not be generated since the ordering restriction incorporated in the algorithm ensures that among those elements generated simultaneously only those elements are *visible* to another element which belong to sets ranked higher than the set containing that element. As an example suppose $S_1(t = 1) = \{\mathtt{a}_1\}$, $S_2(t = 1) = \{\mathtt{b}_1\}$, and $S_3(t = 1) = \{\mathtt{c}_1\}$ and at instant $t = 2$ elements $\mathtt{a}_2$, $\mathtt{b}_2$, and $\mathtt{c}_2$ for the three respective sets are generated simultaneously. Then following the ordering $S_1 < S_2 < S_3$ and considering the three new elements only, both $\mathtt{b}_2$ and $\mathtt{c}_2$ are visible to $\mathtt{a}_2$, while only $\mathtt{c}_2$ is visible to $\mathtt{b}_2$, and nothing is visible to $\mathtt{c}_2$. Thus:

$$PTS(\mathtt{a}_2, t = 2) = \{\langle \mathtt{a}_2, \mathtt{b}_1, \mathtt{c}_1 \rangle, \langle \mathtt{a}_2, \mathtt{b}_1, \mathtt{c}_2 \rangle, \langle \mathtt{a}_2, \mathtt{b}_2, \mathtt{c}_1 \rangle, \langle \mathtt{a}_2, \mathtt{b}_2, \mathtt{c}_2 \rangle\}$$

$$PTS(\mathtt{b}_2, t = 2) = \{\langle \mathtt{a}_1, \mathtt{b}_2, \mathtt{c}_1 \rangle, \langle \mathtt{a}_1, \mathtt{b}_2, \mathtt{c}_2 \rangle\}$$

$$PTS(\mathtt{b}_2, t = 2) = \{\langle \mathtt{a}_1, \mathtt{b}_1, \mathtt{c}_2 \rangle\}$$

Note that at $t = 1$ the cross-product set would be equal to $\{\langle \mathtt{a}_1, \mathtt{b}_1, \mathtt{c}_1 \rangle\}$ and thus adding elements of PTSs of $\mathtt{a}_2, \mathtt{b}_2$, and $\mathtt{c}_2$ to it would generate the correct cross-product set at $t = 2$.

Note that allowing more than one element, which belong to the same set, to have the same time-stamps can lead to race conditions. We therefore assume that exclusive access to a set is obtained when an element is time-stamped and added to it. If there is one lock per set then this will not restrict elements being added to other sets simultaneously.

3.2. Experimental Results

We implemented the time-stamping technique for dynamic cross-product generation on a 10-processor Sequent Symmetry. The elements of sets were generated randomly in a dynamic

fashion. The sets were represented as singly linked list. For simplicity we assumed that all set eventually consisted of an equal number of member elements.

The results of our experiments are presented in table 1. Each row corresponds to a run of the algorithm for a particular problem size. The first column tells the number of sets used for computing the cross-product, the second column the number of elements in each set, the third column the numbers of elements in the resultant cross-product set, the 4th column shows the time taken by an algorithm optimized for sequential execution for that problem size, columns 5th through 10th tell the time taken for cross-product computation using 1, 2, 4, 6, 8, and 10 processors respectively. All the times are in seconds. Figure 1 shows the speed-up graphs for each problem size (the graph for ith row, not counting the heading row, is shown in fig. 1.i). Our results show that the speed-ups obtained for large as well as small sets are linear with respect to the number of processors. For very small sets the speed-ups are sub-linear (fig. 1.4), but the absolute performance is still quite good. The last row in table 1 gives timings for algorithm given in program 2, in which all the sets are locked while a PTS is being computed. As can be observed (fig. 1.6) there is no speedup as more processors are added. Comparing the figures in columns 4th and 5th shows that the performance of the parallel algorithm on one processor is within a factor of 5% compared to the optimized sequential algorithm. This demonstrates that the overhead due to locking etc. is marginal and the technique is indeed quite efficient.

#sets	#ele	#cp-ele	seq	1 P	2 P	4 P	6 P	8 P	10 P
2	1260	1587600	23.2	24.46	12.36	6.22	4.27	3.26	2.58
3	280	21952000	311.38	325.11	162.29	81.25	54.19	40.71	32.63
4	30	810000	11.76	12.31	6.25	3.18	2.21	1.66	1.4
2	360	129600	1.9	1.98	1.05	0.60	0.5	0.43	0.33
2	540	291600	4.21	4.38	2.28	1.20	0.88	0.69	0.57
2	540	291600	4.20	4.21	4.21	4.30	4.41	4.38	4.36

Table 1: Performance Figures

4. Applications

We now briefly discuss some of the applications for which dynamic generation of cross-products is useful.

4.1. And-Or Parallel Execution of Logic Programs

Consider a Prolog goal

```
?- a(X), b(Y), c(X, Y).
```

where sub-goals `a`, `b` and `c` have a number of matching clauses. Let us assume that `a` and `b` are independent of each other, and hence can be executed in and-parallel, and sub-goal `c` is

dependent on `a` and `b`. The various solutions to this goal can be found in parallel by trying to prove sub-goals `a` and `b` in and-parallel and then proving `c`. To prove sub-goals `a, b` and `c` they have to be matched with their respective clauses through unification. This match can be carried out in *or-parallel*. As a result of these or-parallel processes a number of solutions would be produced for both `a` and `b`. Given the semantics of Prolog the solutions to the query would be obtained by cross-producing the solutions set of `a` with that of `b` and then proving sub-goal `c` for each of these combinations. In a parallel system (e.g. [RK89, GJ89]) the solutions to `a` and `b` would be found asynchronously by the run-time system. Thus, the need to dynamically generate the cross-product set arises so that execution of sub-goal `c` can begin as soon as solutions are found for sub-goals `a` and `b`.

4.2. Parallel Evaluation of Data-base Joins

Parallel Evaluation of Joins of Databases has been subject of intense research [QI88, B83, VG84, DG85]. However, these researchers only consider join of static sets, i.e. sets whose size and elements have been completely determined before the join operation is performed. The technique we have developed for parallel evaluation of cross-product sets can also be used in evaluation of join of static sets at a finer grain level. Typically the sets to be joined that arise in databases are very large, and the popular techniques that use parallel processing (for example [B83]) hash the two sets on the join attributes and distributes the different segments that arise to various processors. Given a segment each of the two sets, which map to the same hash-value, the processor matches each element of the segment of the first set with each element of the segment of the second. This last operation is similar to computing the cross-product set of two sets. The matching process can be started as soon as tuples belonging to a segment start arriving at a processor. It is our hope that this dynamic evaluation would add further parallelism to evaluation of database joins and can be incorporated in data-base machines [LM83] to make them even faster.

One can also envisage *hybrid multiprocessor machines*, which consist of a multiple shared memory multiprocessors connected by a message passing network, for evaluating database joins. The join is computed by first distributing the segments which hash to the same value to different shared memory components of the hybrid machine. Within a shared memory component the segments are dynamically cross-produced using the technique presented in this paper to produce the final result of the join.

The technique can also be used in parallel evaluation of relational expressions [G90].

5. Conclusion

We presented an algorithm for parallel dynamic evaluation of the cross-product of a number of dynamically generated sets. The technique is quite simple and uses time-stamps to exploit maximal parallelism. It was implemented on a Sequent Symmetry multi-processor system and linear speed-ups were obtained. The penalty for incorporating time-stamps was

quite negligible, less than 5% of the execution time of the sequential algorithm. Finally, we discussed some applications of dynamic evaluation of cross-product set.

Acknowledgements Thanks to David Warren, Feliks Kluźniak and Bharat Jayaraman for their comments on the manuscript which lead to tremendous improvements. The research was partially supported by U.K. Science and Research Council Grant GR/F 27420.

References

[B83] D. Bitton *et al*, "Parallel Algorithms for the Execution of Relational Database Operations," *ACM Trans. on Database Systems*, **8**,3.

[BG80] P. A. Bernstein and D.W. Shipman, "Time-stamp based algorithms for concurrency control in distributed database systems," In *Proceedings of Conference on Very Large Databases*, 1980, pp. 285-300.

[DG85] D.J. DeWitt and R. Gerber, "Multi-processor hash-based join algorithms," in *Proc. 11th Int. Conf. Very Large Data Bases*, 1985, pp. 151-164.

[G90] G. Gupta, "Dynamic Parallel Evaluation of Cross Product of Sets, TR-90-06, Univ. of Bristol, Apr '90, 14 pp.

[GJ89] G. Gupta and B. Jayaraman, "Compiled And-Or Parallel Execution of Logic Programs," In *Proc. of the North American Conf. in Logic Programming*, MIT Press, Oct. 1989.

[J89] D. W. Jones, "Concurrent operations on Priority Queues," *CACM*, Vol. 32, No. 1, Jan '89.

[LM83] H.-O. Leilich and M. Misskikoff (ed.), "Database Machines," Springer Verlag, 1983.

[QI88] G. Z. Qadah and K. B. Irani, "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine,," in *IEEE Trans. on S/W Engg.*, **14**,11, Nov. 1988.

[RK89] B. Ramkumar and L. V. Kale, "Compiled Execution of the REDUCE-OR Process Model," In *Proc. of North American Conf. on Logic Programming*, MIT Press, 1988.

[U88] J.D. Ullman, "Principles of Data Base and Knowledge Base Systems," Computer Science Press, 1988.

[VG84] P. Valduriez and G. Gardarin, "Join and semijoin algorithms for a multi-microprocessor database machine," *ACM Trans. on Database Systems*, **9**,1.