Abstract—We provide language and compiler support for auto-parallelization of code employing hash-tables. i.e., using our language support, users can write sequential programs employing hash-tables and our compiler will convert that sequential code to a parallel one which runs efficiently on multiple cores or GPUs. The data dependences which may be present in the user code are analyzed by our compiler and based on it, it will try to extract as much parallelism as possible from the user code. The programmer is provided with a set of hash-table functions, which he can use as normal serial functions without any parallelism concerns. Still, the code is able to give the performance of a parallel one.

The auto-parallelism of hash-table is implemented using Pluto- an automatic parallelization tool based on the polyhedral model. This allows the transformed code to benefit from the polyhedral representation of programs. The transformed code generated by Pluto has given ideal performance when parallelized and run on a multi-core system with 12 cores. The transformed code gives a speed up of 6.56x over single thread execution for string counting application and it increases to 10x for counting the permutation of strings. When compared to the manually written TBB code for polynomial multiplication, our code is giving a speed up of 1.27x.

I. INTRODUCTION

Parallel Programming is becoming increasingly common due to the development of multi-core architectures and the need to solve large and complex problems. The need to provide programmers with an easy to use and efficient framework to write parallel programs is a challenging task. Programmers often find it hard to write parallel programs due to the complexity in syntax and the problem of data dependences which can give unexpected results which are also hard to debug. So, its always useful if the compiler can extract parallelism from a sequentially written program. But the task to extract parallelism from a sequentially written code is even more challenging.

Parallelization support for data structures is a key part of parallel programming. In this paper we present a framework through which we can parallelize the code for hash-table, which is one of the most commonly used data structure, due to its ease of use and constant time complexities for usage functions. Hash table has found increased applicability in domains like compilers, search engines, data storage etc. So, parallelization of hash-table usage will benefit all these applications.

We are providing a mechanism by which programmer can write sequential code using hash-table functions and our framework will automatically detect the data dependence in the code and parallelize it. By parallelizing hash-table operations [1], we are parallelizing the user code which contains calls to hash-table functions. The data dependence among hash-table functions are analyzed by our compiler and the sequentially written user code is transformed to a parallel implementation of hash-table, ensuring correctness at the same time increasing the efficiency. The parallelization of hash-table functions also ensures that the other code, which can be done in parallel can now do so.

Also, our work is integrated into Pluto- an automatic parallelization tool[2], based on the polyhedral model[3], thereby exploiting all the optimizations given by the polyhedral representation of programs.

II. RELEVANT WORK

There are existing library support for parallel implementation of data structures including hash-tables. The most common parallel implementations of hash-table are provided by Intel Thread Building Block [4] and STAPL (Standard Template Adaptive Parallel Library)[5].

Threading Building Blocks(TBB) is a C++ template library for parallelism that extends C++ by abstracting away thread management and allowing straightforward parallel programming. To use this library, we have to specify tasks, not threads, and let the library map tasks onto threads in an efficient manner. Threading Building Blocks has a concurrent_hash_map < Key, T, HashCompare >, which is a hash-table that permits concurrent accesses. The table is a map from a key to a type T. The HashCompare traits type defines how to hash a key and how to compare two keys.
STAPL is also a framework for developing parallel programs in C++. It is designed to work on both shared and distributed memory parallel computers. Its core is a library of ISO Standard C++ components with interfaces similar to the (sequential) ISO C++ standard library [6]. STAPL includes a run-time system, design rules for extending the provided library code, and optimization tools. Its goal is to allow the user to work at a high level of abstraction and hide many details specific to parallel programming, to allow a high degree of productivity, portability, and performance. STAPL has a parallel container called pHashMap which is the parallel version of hash_map in Standard Template Library.

Both concurrent_hash_map of TBB and pHashMap of STAPL are framework for users to write parallel programs using hash-tables. The key to their use is that the programmer is freed from the knowledge of using threads. But, even then he has to know how to write the code in a parallel fashion knowing the data dependences. Unlike the library support, we are providing language support for the parallel implementation of hash-table functions. Also, the user is freed from the burden of writing parallel code. He can now write normal sequential code using hash-table functions and our compiler will automatically parallelize that code. We are also implementing our language support over C language and our polyhedral framework is able to generate the TBB code for the parallel execution of hash-table functions.

III. BACKGROUND

A. Polyhedral Model

Polyhedral model [3] is a geometrical as well as linear algebraic framework for capturing the execution of a program for analysis and transformation. It provides an abstraction to perform high-level transformations such as loop-nest optimization and parallelization on affine loop nests. In polyhedral model, the relation between dynamic instances of two loop statements is captured by a dependence polyhedron. From the dependence polyhedron, transformations can be applied to the user code to improve the running time at the same time preserving the dependences.

B. Pluto

PLUTO [2] is an automatic parallelization tool based on the polyhedral model. Pluto transforms an input C program to a code to be run on a multi-core system, exploiting parallelism as well as data locality. The core transformation framework mainly works by finding affine transformations for efficient tiling and fusion, but not limited to those. Outer, inner, or pipelined parallelization is achieved (purely with OpenMP pragmas), besides register tiling and making code amenable to auto-vectorization.

Our work of auto-parallelization of hash-table code is integrated into Pluto. So, now using Pluto we can transform an input C program containing hash-table codes, to a code to be run on a multi-core environment exploiting parallelism as well as data locality. i.e; this work adds the support of hash-tables into the syntax of the C language that Pluto can transform and parallelize.

IV. LANGUAGE SUPPORT

Our hash-table implementation is provided over the C language. So, the following additions needed to be made to the C syntax.

```c
hashtable < datatype, datatype > hashtablename;
```

This, declares a hash-table that maps a key (which can be any data type in C as well as a user defined type) to a value (which can also be any data type).

We are providing the following six hash-table functions:

1. Insert: ht_insert(ht, key, value);

   Inserts a given ⟨key, value⟩ pair to the specified hash-table object ht. If the given key is already present in hashtable ht_insert fails.

2. Delete: ht_delete(ht, key, value);

   Deletes the ⟨key, value⟩ pair from the hash-table ht. If the key is not present in the hash-table ht_delete fails.

3. GetAllKeys: ht_getAllKeys(ht,n);

   Retrieves all the keys stored in the hash-table ht and assigns the total number of keys in ht to n.

4. Modify: ht_modify(ht, key, value);

   Replaces the value corresponding to the specified key from the hash-table ht, if present, with the supplied value. If the key is not already present, it does an insert operation.

5. Update: ht_update(ht, key, operator, value);

   Update operation will update the value corresponding to the key with the new value, which is
oldvalue< operator >operand. The operator here must be associative which enables all the ht_update operations to be run in parallel. This is the major difference between ht_modify and ht_update. If the key is not already there in the hash-table, update will do an insert operation by inserting the default value for that operation. The associative operators being considered are + and * . The default values of these 0 and 1 respectively.

6. Search: ht_search(ht, key);

Searches the hash-table ht for the given key and if found returns the corresponding value or else returns NULL.

V. DATA DEPENDENCE ANALYSIS

The programming control structure on which we focus is [7] loop, because, in general, most of the execution time of a program takes place inside some form of loop. Our framework tries to split up a loop so that its iterations can be executed on separate processors concurrently.

In order to determine whether each iteration of a loop can be executed independent of the others, data dependence analysis needs to be performed for the loops. So, a data dependence representation is also needed. For hash-table functions, the data dependences can be represented as follows:

Two loop statements s1 and s2 are data dependent if any of the following conditions hold

1:

s1 contains ht_insert(ht1, key1, value1)
and
s2 contains ht_delete(ht2, key2, value2)
and
ht1 = ht2
and
key1 = key2.

An insert into a hash-table has a data dependence with a delete from the same hash-table when the key inserted and deleted are the same. If a delete operation precedes an insert and if the hash-table doesn’t already contain that key, then delete will fail. On the other hand, if insert precedes delete, then both will be successful. So, both the operations cannot be done in parallel, if the keys are the same. In all other cases, both the operations can go along simultaneously.

2:

s1 contains ht_insert(ht1, key1, value1)
and
s2 contains ht_search(ht2, key2)
and
ht1 = ht2
and
key1 = key2.

A search operation on a hashtable succeeds only if the key has been previously inserted to the hash-table. So, the insert and search operations cannot be done in parallel if the keys for both the operations are the same.

3:

s1 contains ht_delete(ht1, key1, value1)
and
s2 contains ht_search(ht2, key2)
and
ht1 = ht2
and
key1 = key2.

A search operation on a hashtable fails if the key has been deleted from the hash-table. So, the delete and search operations cannot be done in parallel if the keys for both the operations are the same.

4:

s1 contains ht_insert(ht1, key1, value1)
and
s2 contains ht_update(ht2, key2, operator, value2)
and
ht1 = ht2
and
key1 = key2.

An insert into a hash-table has a data dependence with an update to the same hash-table when the key inserted and updated are the same. If an update operation precedes an insert and if the hash-table doesn’t already contain that key, then the update operation will perform the insert. Then, when the insert operation comes, it’ll fail. So, these two operations cannot be done in parallel, if the keys are the same.

5:

s1 contains ht_insert(ht1, key1, value1)
and
s2 contains ht_modify(ht2, key2, value2)
and
ht1 = ht2
and
key1 = key2.
An insert into a hash-table has a data dependence with a modify to the same hash-table when the key inserted and updated are the same. If a modify operation precedes an insert and if the hash-table doesn’t already contain that key, then the modify operation will perform the insert. Then, when the insert operation comes, it’ll fail. So, these two operations cannot be done in parallel, if the keys are the same.

6:

\[
\begin{align*}
s1 & \text{ contains } \text{ht}\_\text{search}(ht1, key1) \\
& \text{ and } \\
s2 & \text{ contains } \text{ht}\_\text{update}(ht2, key2, value2) \\
& \text{ and } \\
& \text{ht1} = \text{ht2} \\
& \text{ and } \\
& \text{key1} = \text{key2}.
\end{align*}
\]

A modify and an update operation cannot be done in parallel for the same key. If the search operation is after an update operation, it’s expected to return the updated value. But if the search operation happens to execute before the modify operation in the parallel run, it will return the old value. So, these two operations cannot be parallelized.

7:

\[
\begin{align*}
s1 & \text{ contains } \text{ht}\_\text{insert}(ht1, key1, value1) \\
& \text{ and } \\
s2 & \text{ contains } \text{ht}\_\text{getAllKeys}(ht2) \\
& \text{ and } \\
& \text{ht1} = \text{ht2}.
\end{align*}
\]

ht\_getAllKeys will retrieve all the keys present in the hash-table. So, an insert operation cannot be done in parallel with it since it can modify the set of keys in the hash-table.

8:

\[
\begin{align*}
s1 & \text{ contains } \text{ht}\_\text{delete}(ht1, key1, value1) \\
& \text{ and } \\
s2 & \text{ contains } \text{ht}\_\text{getAllKeys}(ht2) \\
& \text{ and } \\
& \text{ht1} = \text{ht2}.
\end{align*}
\]

ht\_getAllKeys will retrieve all the keys present in the hash-table. So, similar to insert, delete operation also cannot be done in parallel with it since it also can modify the set of keys in the hash-table.

9:

\[
\begin{align*}
s1 & \text{ contains } \text{ht}\_\text{modify}(ht1, key1, value1) \\
& \text{ and } \\
s2 & \text{ contains } \text{ht}\_\text{modify}(ht2, key2, value2) \\
& \text{ and } \\
& \text{ht1} = \text{ht2} \\
& \text{ and } \\
& \text{key1} = \text{key2}.
\end{align*}
\]

A modify operation will change the value corresponding to a key if that key is already present in the hashtable. So, two modify operations cannot be done in parallel since we cannot assure which value the key will contain after the parallel operations.

10:

\[
\begin{align*}
s1 & \text{ contains } \text{ht}\_\text{delete}(ht1, key1, value1) \\
& \text{ and } \\
s2 & \text{ contains } \text{ht}\_\text{modify}(ht2, key2, value2) \\
& \text{ and } \\
& \text{ht1} = \text{ht2} \\
& \text{ and } \\
& \text{key1} = \text{key2}.
\end{align*}
\]

A modify and a delete operation cannot be done in parallel for the same key. If delete is done before modify, the effects of delete will be removed by the modify operation and if delete is done after the modify operation, the effects of modify will be removed by the delete operation.

11:

\[
\begin{align*}
s1 & \text{ contains } \text{ht}\_\text{search}(ht1, key1) \\
& \text{ and } \\
s2 & \text{ contains } \text{ht}\_\text{modify}(ht2, key2, value2) \\
& \text{ and } \\
& \text{ht1} = \text{ht2} \\
& \text{ and } \\
& \text{key1} = \text{key2}.
\end{align*}
\]

A modify and a search operation cannot be done in parallel for the same key. If a search operation is after the modify operation, its expected to return the modified value. But if the search operation happens to execute before the modify operation, it will return the old value. So, these two operations cannot be parallelized.

12:

\[
\begin{align*}
s1 & \text{ contains } \text{ht}\_\text{modify}(ht1, key1, value1) \\
& \text{ and } \\
s2 & \text{ contains } \text{ht}\_\text{getAllKeys}(ht2) \\
& \text{ and } \\
& \text{ht1} = \text{ht2}.
\end{align*}
\]

If a key is not present in the hash-table, modify operation will work similar to an insert operation. So, ht\_modify
and ht_getAllKeys cannot be done in parallel for a hash-table object.

13:

\[ s1 \text{ contains } \text{ht_update}(ht, \text{key}, \text{operator}, \text{value}) \]
\[ \text{and} \]
\[ s2 \text{ contains } \text{ht_delete}(ht1, \text{key1}, \text{value1}) \]
\[ \text{and} \]
\[ ht1 = ht2 \]
\[ \text{and} \]
\[ \text{key1} = \text{key2}. \]

If a key is not present in the hash-table, update operation will work similar to an insert operation. So, ht_update and ht_delete cannot be done in parallel for the same key.

14:

\[ s1 \text{ contains } \text{ht_update}(ht, \text{key}, \text{operator}, \text{value}) \]
\[ \text{and} \]
\[ s2 \text{ contains } \text{ht_getAllKeys}(ht2) \]
\[ \text{and} \]
\[ ht1 = ht2 \]

Even though update operation modifies the value only, if the key is not already present in the hash-table, it will do an insert to the hash-table. So, it cannot be done in parallel with getAllKeys method if the updated key is not already present in the hash-table.

15:

\[ s1 \text{ contains } \text{ht_update}(ht, \text{key}, \text{operator}, \text{value}) \]
\[ \text{and} \]
\[ s2 \text{ contains } \text{ht_modify}(ht2, \text{key2}, \text{value2}) \]
\[ \text{and} \]
\[ ht1 = ht2 \]
\[ \text{and} \]
\[ \text{key1} = \text{key2}. \]

The update operation operates on the value corresponding to the key and the modify operation replaces the value corresponding to the key. So, these two operations cannot be done in parallel for the same key as the execution order can change the final value corresponding to the key.

The dependences among the hash-table functions are shown in Figure 1. The \( X \) mark shows the functions which cannot be parallelized. i.e., the functions between which there is a data dependence. A \( \text{X} \) shows that there is no data dependence between two hash-table functions and hence marks the functions which are parallel. For example, the \( \text{X} \) mark for Insert and GetAllKeys means that there is a data dependence between these two and hence the order of these statements in the user code cannot be changed. Similarly, the \( \text{X} \) between Update and Update means that two Update operations can be done in parallel.

### VI. IMPLEMENTATION

#### A. Dependence Representation

For each hash-table, a state is assigned which shows what all functions have been called on that hash-table in a particular loop body. Table 1 shows this state information. If the state of a hash-table upon exit from a loop body is such that there are no data dependence in that loop, then all the function calls to that hash-table in that loop can be run in parallel. If the states of all the hash-tables upon exit from a loop, is such that there are no data dependence among them then all the iterations of that loop can be run in parallel.

Table 1 shows the six possible hash-table states. Out of this, the first five are valid states for parallelization, i.e., if a hash-table is in any of these six states upon exit from a loop body, then all the accesses to that table in that loop can be run in parallel.

#### B. Hash Table Implementation

For parallel hash-table implementation we are using Intel Thread Building Block(TBB). TBB has a concurrent_hash_map data type which supports parallel implementation of hash-table functions. A concurrent_hash_map acts as a container of elements of type \( \text{std}::\text{pair} < \text{constKey}, T > \). Typically, when accessing a container element, we are interested in either updating it or reading it. The template class concurrent_hash_map supports these two operations with the accessor and const_accessor classes, respectively, which act as smart pointers.

An accessor represents update (write) access. As long as it points to an element, all other attempts to look up that key in the table block until the accessor is done. A const_accessor is similar, except that it represents read-only access. Therefore, multiple const_accessors can point to the same element at the same time. This feature can
greatly improve concurrency in situations where elements are frequently read and infrequently updated.

The find and insert methods take an accessor or const accessor as an argument. The choice tells concurrent_hash_map whether we are asking for update or read-only access, respectively. Once the method returns, the access lasts until the accessor or const_accessor is destroyed.

C. Modifications to Pluto

Pluto uses Clan[8] to extract the polyhedral representation from user programs. Clan has been modified to add support for hash-tables. New parsing rules have been added for hash-table declarations as well as the six hash-table function calls. The scopplib structure which Pluto uses for storing the polyhedral information has also been modified to add entry for the list of hash-table names and the hash-table state informations. Support has also been added to Cloog[9] - used to generate the transformed code by Pluto, to generate the code for hash-table.

D. Transformations to Hash Table Functions

1. ht_insert(table, key, value)
   Transformed Code:
   
   ```cpp
   table.insert(std::make_pair(key, value));
   ```

2. ht_delete(table, key)
   Transformed Code:
   
   ```cpp
   table.erase(key);
   ```

3. ptr = ht_search(table, key)
   Transformed Code:
   
   ```cpp
   HashTable::const_accessor acc;
   if (table.find(acc, key))
      ptr = acc->second;
   else
      ptr = 0;
   ```

4. ptr = ht_getAllKeys(table, key, value)
   Transformed Code:
   
   ```cpp
   for (HashTable::iterator i=table.begin(); i!=table.end(); ++i )
   {
      ptr = i->first;
      ptr++;
   }
   ```

5. ht_modify(table, key, value)
   Transformed Code:
   
   ```cpp
   HashTable::accessor acc;
   table.insert(std::make_pair(key, 0));
   if (table.find(acc, key))
      acc->second = value;
   ```

6. ht_update(table, key, operator, value)
   Transformed Code:
<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>12</td>
</tr>
<tr>
<td>Clock speed</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>24 GB</td>
</tr>
<tr>
<td>Linux Version</td>
<td>2.6.32</td>
</tr>
<tr>
<td>gcc Version</td>
<td>4.4.4</td>
</tr>
</tbody>
</table>

**TABLE II: System Configuration**

```c
// for '+' operator
HashTable::accessor acc;
table.insert(std::make_pair(key, 0));
if (table.find(acc, key))
  acc->second += value;

// for '*' operator
HashTable::accessor acc;
table.insert(std::make_pair(key, 1));
if (table.find(acc, key))
  acc->second *= value;
```

**E. Further Optimization**

By default a for loop containing a modify call to a hash-table cannot be parallelized. But this can be done if it is sure that the keys to each modify call in the loop is different. Using this information, a for loop can be parallelized. For example if the keys being used in the modify calls in a for loop are distinct elements of an array and if that array is a set of elements (possible returned by a call to getAllKeys), then its sure at compile time that the keys to different hash-table calls are different and hence can be parallelized.

The information that an array contains only distinct elements is added to the scoplib structure being created by CLAN[8]. This information is used by Pluto to determine if a modify call in a for loop can be parallelized.

**VII. EXPERIMENTAL EVALUATION**

The experimental codes were run on the system with configurations as shown in Table 1.

**A. String Count**

Here, we are showing how we are transforming a serial code written using our hash-table functions into a parallel code to be run using Intel Thread Building Block. The code written is for counting the number of occurrences of each word in a list of words. The code is written inside #pragma scop, for CLAN to extract the polyhedral information from it. In the code, each word (stored in the array `data`) acts as a key in the hash-table and the value corresponds to the number of occurrences of that word in the list of words.

```c
#pragma scop
for (j = 0; j < n; j++) {
  ht_update(table, data[j], '+', 1);
}
#pragma endscop
```

1) Applying Dependences: Here, there is only one hash-table object in the user code. There is a for loop for inserting words into the hash-table object `table` and there is only one hash-table function `ht_update` inside the loop. As per the dependence relation, there is no data dependence between a `ht_update` and an other. So, all the iterations of the for loop can be done in parallel.

2) Transformed Code: The following is the transformed code for the above program.

```c
/* Start of CLooG code */
/* Generated from PLUTO-produced CLooG file by CLooG 0.16.3-UNKNOWN gmp bits in 0.00s. */
if (i >= 1) {
  for (t1=0; t1<=i-1; t1++) {
    StringTable::accessor acc;
    table.insert(std::make_pair(data[t1], 0));
    if (table.find(acc, data[t1]))
      acc->second += 1;
  }
}
/* End of CLooG code */
```

In the transformed code an accessor is used to insert strings to the hash-table. This is because we are writing to the hash-table and so we need a write lock on the particular location where we are writing, so that no other operations on that key can be performed until the lock is released. But, at the same time operations are allowed for other keys in the hash-table ensuring parallelism.

Figure 2 plots the execution time of the transformed code run using TBB, when the number of threads was varied. The experiment was run for strings of number 1,000,000. The execution time was best when the number of threads was twelve and then it became constant even when the number of threads was increased. This was expected as the experiment was run on a 12 core machine; more than 12 threads means a thread has to wait for CPU from other threads.

**B. Increasing the work per thread**

The following code counts the number of occurrence of permutations of each word in a list of words. The call classify sorts the string passed to it, so that all permutations of words would have identical key value. So, in each iteration the amount of work is increased.

```c
#pragma scop
for (j = 0; j < i; j++) {
    classify(data[j]);
    ht_update(table, data[j], '+' , 1);
}
#pragma endscop
```

Figure 3 plots the execution time of the transformed code with increased load per iteration. As expected, this program took more time compared to the previous one but gave an improved performance when the number of threads was increased.

**C. Benefiting from Polyhedral Representation**

The following code is for multiplying two polynomials storing the result in the hash-table (key is the degree of the term and value is its co-efficient).

```c
#pragma scop
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        c[i+j] = c[i+j] + a[i]*b[j];
        ht_update(poly,add(i,j),+ ,mul(a[i],b[j]));
    }
}
#pragma endscop
```

The transformed code for the above one is in Appendix.

Figure 4 shows the performance comparison of Pluto generated code and the manually written Intel TBB code. Pluto generated code is giving better performance compared to the manually written Intel TBB code. Also, Pluto generated code is more scalable to parallelism- giving better performance when the number of threads is large.
VIII. CONCLUSION AND FUTURE WORK

A framework for the users to write sequential programs using hash-table functions has been developed. From the input user code, CLAN will extract the hash-table details. From the extracted details Pluto is now able to detect the data dependences among the hash-table accesses. Depending on the data dependence information Pluto is able to generate the suitable code to be run on multi-core systems exploiting features such as data locality. The transformed code generated by Pluto has given ideal performance when parallelized and run on a multi-core system. The transformed code gives a speed up of 6.56x over single thread execution for string counting application and it increases to 10x for counting the permutation of strings.

When compared to the manually written TBB code for polynomial multiplication, our code is giving a speed up of 1.27x.

Currently, the parallel code generated by Pluto for hash-tables can be run only on multi-core systems using shared memory. This can be extended to distributed systems using MPI framework. Also, similar to hash-table other dynamic data structures can be added to Pluto exploiting the benefits of auto-parallelism and polyhedral representation.

REFERENCES


APPENDIX

Polynomial Multiplication Code - Transformed by Pluto
/* Start of CLooG code */
/* Generated from PLUTO-produced CLooG file by CLooG 0.16.3—UNKNOWN gmp bits in 0.23s. */

if (n >= 1) {
    lb1=0;
    ub1=floord(n-1,16);
    #pragma omp parallel for shared(lb1,ub1)
        private(ubv,lbv,t1,t2,t3,t4)
        for (t1=lb1; t1<=ub1; t1++) {
            for (t2=max(0,ceild(32*t1-n-30,32)); t2<=floord(n-1,32);t2++) {
                if (((t1 <= floord(n-1,32)) & & (t2 == 0)) {

                    lbv=32*t1; ubv=min(n-1,32*t1+31);
                    for (t4=lbv; t4<=ubv; t4++) {
                        c[0 +t4]=c[0 +t4]+a[0]*b[t4];
                        StringTable::accessor acc;
                        poly.insert(std::make_pair(add(t4,0),
                            if (poly.find(acc, add(t4,0)))
                                acc -> second += mul(a[t4],b[0]);
                    }
                } else {
                    if (((t1 <= floord(n-1,32)) &&
                        (t1 >= ceild(n-31,32))) {
                        for (t3=max(1,32*t2);
                            t3<=min(32*t1,32*t2+31);t3++)
                        {
                            lbv=32*t1; ubv=n-1;
                            for (t4=lbv; t4<=ubv; t4++) {
                                c[t3+t3+t4]=c[t3+t3+t4]+a[t3]*b[-t3+t4];
                                StringTable::accessor acc;
                                poly.insert(std::make_pair(add(t4,t3),
                                    if (poly.find(acc, add(t4,t3)))
                                        acc -> second += mul(a[t4],b[t3]);
                                } else {
                                    lbv=n; ubv=min(32*t1+31,t3+n-1);
                                    for (t4=lbv; t4<=ubv; t4++) {
                                        c[t3+t3+t4]=c[t3+t3+t4]+a[t3]*b[-t3+t4];
                                    }
                }
            }
        }
    }
}