Language and compiler parallelization support for Hash tables

Arjun Suresh
Advisor: Dr. UDAY KUMAR REDDY B.

Department of Computer Science & Automation
Indian Institute of Science, Bangalore
Bengaluru, India.
We provide language and compiler support for auto-parallelization of code employing hash tables.

The programmer is provided with a set of hash table functions, which he can use as normal serial functions without any parallelism concerns.

The data dependences are analyzed by our compiler.

It will try to extract as much parallelism as possible from the user code.

The work is integrated into PLUTO, an automatic parallelization tool based on the polyhedral model.
Overview

- We provide language and compiler support for auto-parallelization of code employing hash tables
- The programmer is provided with a set of hash table functions, which he can use as normal serial functions without any parallelism concerns
- The data dependences are analyzed by our compiler
- It will try to extract as much parallelism as possible from the user code
- The work is integrated into PLUTO, an automatic parallelization tool based on the polyhedral model
We provide language and compiler support for auto-parallelization of code employing hash tables. The programmer is provided with a set of hash table functions, which he can use as normal serial functions without any parallelism concerns. The data dependences are analyzed by our compiler. It will try to extract as much parallelism as possible from the user code. The work is integrated into PLUTO, an automatic parallelization tool based on the polyhedral model.
Overview

- We provide language and compiler support for auto-parallelization of code employing hash tables.
- The programmer is provided with a set of hash table functions, which he can use as normal serial functions without any parallelism concerns.
- The data dependences are analyzed by our compiler.
- It will try to extract as much parallelism as possible from the user code.
- The work is integrated into PLUTO, an automatic parallelization tool based on the polyhedral model.
We provide language and compiler support for auto-parallelization of code employing hash tables.

The programmer is provided with a set of hash table functions, which he can use as normal serial functions without any parallelism concerns.

The data dependences are analyzed by our compiler.

It will try to extract as much parallelism as possible from the user code.

The work is integrated into PLUTO, an automatic parallelization tool based on the polyhedral model.
Parallel Programming becoming increasingly common

- Parallel programs are hard to write
  - Complexity in syntax
  - Problem of data dependences
  - Different programs needs to be written for muti-core, multiprocessor and GPU environments

- Useful if the compiler can extract parallelism from a sequentially written program and generate code as per the running architecture
Introduction

- Parallel Programming becoming increasingly common
- Parallel programs are hard to write
  - Complexity in syntax
  - Problem of data dependences
  - Different programs need to be written for multi-core, multiprocessor and GPU environments
- Useful if the compiler can extract parallelism from a sequentially written program and generate code as per the running architecture
Parallel Programming becoming increasingly common
Parallel programs are hard to write
  - Complexity in syntax
  - Problem of data dependences
  - Different programs need to be written for multi-core, multiprocessor and GPU environments
Useful if the compiler can extract parallelism from a sequentially written program and generate code as per the running architecture
Parallel Programming becoming increasingly common

Parallel programs are hard to write
- Complexity in syntax
- Problem of data dependences
- Different programs needs to be written for multi-core, multiprocessor and GPU environments

Useful if the compiler can extract parallelism from a sequentially written program and generate code as per the running architecture
Parallel Programming becoming increasingly common
Parallel programs are hard to write
- Complexity in syntax
- Problem of data dependences
- Different programs needs to be written for muti-core, multiprocessor and GPU environments

Useful if the compiler can extract parallelism from a sequentially written program and generate code as per the running architecture
There are parallel library implementations of hash table. The two most common among them are:

- STAPL [1]
- Intel Thread Building Block [2]

Differences from library implementations:

- We are providing a language support for parallel implementation of hash table rather than a library support.
- The data dependence analysis is taken care of by our compiler.
- The user has to just write a sequential code employing hash table.
- The sequential code will be mapped to the suitable parallel code.
PLUTO is an automatic parallelization tool for polyhedral model.

It provides an abstraction to perform high-level transformations such as loop-nest optimization and parallelization on affine loop nests.

PLUTO is based on various polyhedral tools (CLAN, CANDL, and CLOOG) and it generates OPEN MP and MPI codes.
Extending Pluto for Adding Support for Hash Tables

- Add syntax for declaring and accessing a hash table
- Modify CLAN
- Analyze and represent the data dependence in hash table access functions
- Translate the user code to suitable hash table implementations for multicore
Language Support for HashTables

Addition in Syntax:

hashtable < datatype, datatype > hashtablename ;

We are providing six hash table functions:

- Insert: ht_insert(ht, key, value);
- Delete: ht_delete(ht, key, value);
- GetAllKeys: ht_getAllKeys(ht, &n);
- Modify: ht_modify(ht, key, value);
- Update: ht_update(ht, key, operator, value);
- Search: ht_search(ht, key);
Language Support for HashTables

Addition in Syntax:

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

We are providing six hash table functions:

- Insert: `ht_insert(ht, key, value);`
- Delete: `ht_delete(ht, key, value);`
- GetAllKeys: `ht_getAllKeys(ht, &n);`
- Modify: `ht_modify(ht, key, value);`
- Update: `ht_update(ht, key, operator, value);`
- Search: `ht_search(ht, key);`
Language Support for HashTables

Addition in Syntax:

\[
\text{hashtable} \left< \text{datatype}, \text{datatype} \right> \text{hashtablename} ;
\]

We are providing six hash table functions:

- Insert: `ht_insert(ht, key, value);`
- Delete: `ht_delete(ht, key, value);`
- GetAllKeys: `ht_getAllKeys(ht, &n);`
- Modify: `ht_modify(ht, key, value);`
- Update: `ht_update(ht, key, operator, value);`
- Search: `ht_search(ht, key);`
Addition in Syntax:

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

We are providing six hash table functions:

- Insert: `ht_insert(ht, key, value);`
- Delete: `ht_delete(ht, key, value);`
- GetAllKeys: `ht_getAllKeys(ht, &n);`
- Modify: `ht_modify(ht, key, value);`
- Update: `ht_update(ht, key, operator, value);`
- Search: `ht_search(ht, key);`
Addition in Syntax:

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

We are providing six hash table functions:

- Insert: `ht_insert(ht, key, value);`
- Delete: `ht_delete(ht, key, value);`
- GetAllKeys: `ht_getAllKeys(ht, &n);`
- Modify: `ht_modify(ht, key, value);`
- Update: `ht_update(ht, key, operator, value);`
- Search: `ht_search(ht, key);`
Language Support for HashTables

Addition in Syntax:

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

We are providing six hash table functions:

- Insert: `ht_insert(ht, key, value);`
- Delete: `ht_delete(ht, key, value);`
- GetAllKeys: `ht_getAllKeys(ht, &n);`
- Modify: `ht_modify(ht, key, value);`
- Update: `ht_update(ht, key, operator, value);`
- Search: `ht_search(ht, key);`
A data dependence exists from statement $s_1$ to $s_2$ iff,

- Both $s_1$ and $s_1$ access the same memory location and
- At least one of them is a write access
Two loop statements s1 and s2 are data dependent if any of the following conditions hold

- s1 contains \texttt{ht\_insert(ht1, key1, value1)}
  and
- s2 contains \texttt{ht\_delete(ht2, key2, value2)}
  and
- ht1 = ht2
  and
- key1 = key2
Data Dependence Analysis

- $s_1$ contains $\text{ht\_insert}(ht_1, \text{key1}, \text{value1})$
  and
  $s_2$ contains $\text{ht\_search}(ht_2, \text{key2})$
  and
  $ht_1 = ht_2$
  and
  $\text{key1} = \text{key2}$

- $s_1$ contains $\text{ht\_delete}(ht_1, \text{key1}, \text{value1})$
  and
  $s_2$ contains $\text{ht\_search}(ht_2, \text{key2})$
  and
  $ht_1 = ht_2$
  and
  $\text{key1} = \text{key2}$
Data Dependence Analysis

- s1 contains `ht_insert(ht1, key1, value1)` and
  s2 contains `ht_getAllKeys(ht2)` and
  \[ht1 = ht2\]
- s1 contains `ht_delete(ht1, key1, value1)` and
  s2 contains `ht_getAllKeys(ht2)` and
  \[ht1 = ht2\]
Data Dependence Analysis

- s1 contains \( \text{ht\_modify}(ht1, key1, value1) \)
  and
s2 contains \( \text{ht\_modify}(ht2, key2, value2) \)
  and
  \( ht1 = ht2 \)
  and
  \( key1 = key2 \)

- s1 contains \( \text{ht\_delete}(ht1, key1, value1) \)
  and
s2 contains \( \text{ht\_modify}(ht2, key2, value2) \)
  and
  \( ht1 = ht2 \)
  and
  \( key1 = key2 \)
Data Dependence Analysis

- s1 contains `ht_search(ht1, key1)`
  
-and

- s2 contains `ht_modify(ht2, key2, value2)`
  
-and

  `ht1 = ht2`

-and

  `key1 = key2`

- s1 contains `ht_modify(ht1, key1, value1)`
  
-and

- s2 contains `ht_getAllKeys(ht2)`
  
-and

  `ht1 = ht2`
Data Dependence Analysis

- s1 contains `ht_update(ht, key, operator, value)`
  and
- s2 contains `ht_delete(ht1, key1, value1)`
  and
  \[ ht1 = ht2 \]
  and
  \[ key1 = key2 \]
- s1 contains `ht_update(ht, key, operator, value)`
  and
- s2 contains `ht_getAllKeys(ht2)`
  and
  \[ ht1 = ht2 \]
  and
  \[ ht_search(ht, key) == NULL \]
s1 contains \texttt{ht\_update(ht, key, operator, value)}
and
s2 contains \texttt{ht\_modify(ht2, key2, value2)}
and
ht1 = ht2
and
key1 = key2
## Dependences among Hash-table functions

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Search</th>
<th>GetAllKeys</th>
<th>Delete</th>
<th>Modify</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Search</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>GetAllKeys</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Delete</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Modify</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Update</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>
Multiple iterations of all the hash table functions except modify can be run in parallel.

Except, search and getallkeys, no other permutations of hash table functions can be scheduled in parallel unless the compiler can determine that the keys used are different for both the hash table accesses.

A state is assigned to each hashtable which stores information as to which all functions have been called for that hashtable inside a particular loop body.
### HasTable States

<table>
<thead>
<tr>
<th>State</th>
<th>Functions being Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Single or Multiple calls to Insert</td>
</tr>
<tr>
<td>2</td>
<td>Single or Multiple calls to Delete</td>
</tr>
<tr>
<td>3</td>
<td>Single or multiple calls to Search or GetAllKeys</td>
</tr>
<tr>
<td>4</td>
<td>Single call to Modify</td>
</tr>
<tr>
<td>5</td>
<td>Single or Multiple Call to Update</td>
</tr>
<tr>
<td>6</td>
<td>All other combinations of function calls leads to this state</td>
</tr>
</tbody>
</table>

**Table:** HasTable States
Hash Table Implementation for multi-cores

- Makes use of concurrent_hash_map in Intel Thread Building Block
- Supports parallel implementation of hash table
- Accessor and const_accessor classes are used to access hash table elements
- Accessor class is used for write access and const_accessor for read access
Experimental Results

The experimental codes were run on the system with configurations as shown below

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Xeon</td>
</tr>
<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: System Configuration
The code for counting the number of occurrences of each word in a list of words.

```c
#pragma scop
    for (j = 0; j < n; j++) {
        ht_update(table, data[j], '+', 1);
    }
#pragma endscop
```
\begin{lstlisting}
/* Start of CLooG code */
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 */
\end{lstlisting}
Performance of the code for Varying Number of threads

![Bar chart showing execution time for varying number of threads.](chart.png)
The code for counting the number of occurrences of permutations of each word in a list of words.

```c
#pragma scop
    for (j = 0; j < n; j++) {
        classify(data[j]);
        ht_update(table, data[j], '+', 1);
    }
#pragma endscop
```
Performance of the code for Varying Number of threads

![Bar chart showing execution time for varying number of threads.](chart.png)
The following code is for multiplying two polynomials storing the result in a 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
```
Performance of Pluto generated TBB code vs manual TBB code

Pluto Generated TBB Code vs TBB Code

- Execution Time (seconds)
- No. of Threads

<table>
<thead>
<tr>
<th>No. of Threads</th>
<th>TBB Code</th>
<th>Pluto Generated TBB Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4000</td>
<td>4000</td>
</tr>
<tr>
<td>10</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>30</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>40</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>50</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>
A framework for the users to write sequential programs using hash table functions has been developed.

Data dependences between the various hash table functions has been identified and represented inside Pluto.

Transformation of user code to a code to be run on Intel Thread Building Block framework.

Experimental Results have given good results:
- Count Strings application gave a speedup of $6.56 \times$ compared to single thread execution on a 12 core machine.
- Permutation of Strings Count gave a speed up of $10 \times$.
- Polynomial Multiplication gave a speed up of $1.27 \times$ compared to manually written TBB code.
Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger.

The stapl parallel container framework.

James Reinders.

*Intel threading building blocks - outfitting C++ for multi-core processor parallelism.*
Thank You !!!