The handle http://hdl.handle.net/1887/25180 holds various files of this Leiden University dissertation

Author: Rietveld, K.F.D.
Title: A versatile tuple-based optimization framework
Issue Date: 2014-04-10
CHAPTER 12

Controlling Distributed Execution of Forelem Loops

12.1 Introduction

This chapter presents the results of a preliminary investigation into the suitability of the forelem framework for the automatic parallelization of database applications or automatic generation and optimization of Big Data applications for execution on multiple compute nodes. As has been discussed in this thesis, the forelem framework introduces a universal approach for the optimization of an application’s data layout and storage. By incorporating details about the data access performed by the application into the optimization process, the application and its data access method can be synchronized. This synchronization leads to a better alignment of the application’s computational loops with the order in which data is accessed. This can even lead to changes in the storage layout and format of the application. In this chapter extensions to the forelem framework are described so that data distribution can be explicitly controlled in addition to (local) data layout.

The approach that has been chosen for these extensions relies on the techniques derived from the optimization of program code and data distribution to map program codes onto parallel computers, see for instance [51]. In the forelem framework the distribution of data is being handled by special loop constructs which express parallel execution coupled with data decomposition. This data decomposition can be specified by an “automatic” decomposition of the value range of one or more particular fields in the database model. This approach is very generic allowing multiple data decompositions to be considered at compile time.

Big Data applications can also take advantage of the ability of the forelem framework to support vertical integration of application code and data access frameworks. Similar to regular database applications, Big Data applications typically access data through a framework that abstracts away peculiarities of accessing a particular file format, database system or distributed file system. Such frameworks inhibit optimizing compilers from potentially optimizing data access as
performed by an application. Through the use of the forelem framework, the data access operations that are performed through a data access framework are expressed in the generic intermediate representation, unlocking many more potential optimization opportunities.

This chapter takes the following approach: first a scheme is described to influence the distribution of forelem loops and the selection of a data distribution. Secondly, an important transformation in the context of distributed applications, Iteration Space Expansion, is described in detail. Thirdly, it is demonstrated how this scheme can be used together with the compiler optimization techniques that are implemented in the forelem framework to automatically parallelize database applications and generate Big Data codes. On a set of loops parallelized using a data distribution based on the value range of one of the fields, we demonstrate a number of transformations that optimize for the re-use of a selected data distribution. As a consequence, the forelem framework has the power to automatically distribute program codes similar to MapReduce-style computations. In other words, optimizing compiler technology is enabled to be used in the optimization and parallelization of database applications.

The viability of the forelem framework to be used for the optimization of Big Data applications is illustrated using two typical MapReduce examples. We show how from a problem expressed in SQL, a forelem intermediate representation of the problem can be derived and be subsequently optimized through the application of transformations present in the forelem framework. From this intermediate representation, different codes can be generated using different data layouts. Initial experiments show the importance of considering a good data layout for the problem at hand. Implementations generated with the forelem framework realize a performance improvement of a factor 3 compared to a Hadoop implementation when the same input data file is used. Performance improvements up to a factor 120 can be reached if the input data with an optimized layout is automatically generated.

This chapter is organized as follows Section 12.2 describes the extensions to the forelem framework for expressing distributed execution of forelem loops. Section 12.3 discusses the Iteration Space Expansion transformation. Section 12.4 illustrates how the proposed extension and the transformations in the forelem framework are used together and reinforce each other. Section 12.5 illustrates the viability of the forelem framework to optimize Big Data Applications. Section 12.6 discusses the conclusions.

## 12.2 Distribution of Forelem Loops

In Chapter 11 we used the observation that since forelem loops may iterate their index set in any order, forelem loops are inherently parallel. So, in fact, no special semantics are needed to be able to execute a forelem loop in parallel. However, when execution of a forelem loop is to be distributed to multiple nodes, control over what parts of the forelem loop is executed on what node is necessary to be able to optimize data decomposition and distribution. In the remainder of this
chapter, with “parallel execution”, we mean parallel execution distributed over multiple processors (or compute nodes).

Within the forelem framework, parallelization consists out of loop scheduling, which is the problem of scheduling a parallel loop’s iterations onto the available processors, and secondly out of data distribution (or decomposition) to the processors. It is assumed that accesses to data that is not available locally are resolved by performing remote communication to a processor that does have the necessary data available. Loop scheduling is implemented through the application of Loop Blocking to the iteration space of a forelem loop. A distinction is made between direct and indirect loop scheduling, both of which will be described in this section. Finally, it is shown how the data set can be decomposed based on the created loop schedule.

Loop scheduling follows from the application of Loop Blocking to the iteration space of a forelem loop. With direct loop scheduling, the iteration space is blocked by partitioning the index set that is iterated by the forelem loop. On the other hand, indirect loop scheduling is achieved by blocking on the value range of a field in the accessed array.

As an example, consider an array $A$ with fields $field1$ and $field2$, and the following loop where $SEQ$ denotes a sequence of statements:

```plaintext
forelem (i; i ∈ pA)
SEQ;
```

In order to parallelize this loop to $N$ processors, a loop schedule must be created. To create a direct loop schedule, Loop Blocking splits the iteration space of this loop, which is the index set $pA$, into $N$ partitions:

$$pA = p1A \cup p2A \cup \ldots \cup pNA$$

and the forelem loop becomes:

```plaintext
for (k = 1; k <= N; k++)
    forelem (i; i ∈ pkA)
SEQ;
```

Subsequently, to parallelize this loop to $N$ processors, each processor must be assigned a partition of the index set $pA$. This is achieved by replacing the for loop with a forall loop, indicating that the outer loop is executed in parallel:

```plaintext
forall (k = 1; k <= N; k++)
    forelem (i; i ∈ pkA)
SEQ;
```

As a next step, the data can be decomposed according to the selected partitioning. So, a decomposition of table $A$ is created:

$$A = A1 \cup A2 \cup \ldots \cup AN$$

based on the partitioned index sets $pkA$. Note that, this decomposition of $A$ yields an index set $pA_k$ for every $A_k$. The loop operating on the decomposed data is:
forall (k = 1; k <= N; k++)
  forelem (i; i ∈ pA_k)
  SEQ;

where in the loop body data is accessed through for example A_k[i].field1. Note
that, in case data accesses are performed to data that is not available locally af-
ter the data decomposition, these accesses can be resolved by performing remote
communication to a processor that does have the necessary data available.

In indirect data partitioning, Loop Blocking is not done based on the iterated
index set, but on the value range of one of the table’s accessed fields. Consider the
same starting point:

forelem (i; i ∈ pA)
  SEQ;

Array A is to be distributed into N partitions based on field1. The notation A.field1
denotes the set of values of the field1 found in all subscripts of A. If X = A.field1,
then

X = X_1 ∪ X_2 ∪ ... ∪ X_N

is a partitioning of X into N segments. The blocked loop is:

for (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA.field1[l])
    SEQ;

In this loop nest the outer loop can be parallelized. In the parallelized loop nest
a processor P_k is responsible for processing partition X_k of this partitioning and
will execute the original forelem loop only for i ∈ pA, l ∈ X_k : A[i].field1 = l. This
results in:

forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA.field1[l])
    SEQ;

Also in this case, the table A can be decomposed based on the selected indirect loop
schedule. The decomposition of A into N parts A_k, with corresponding index sets
pA_k is based on the partitioning X into X_k. This results in the following loop nest:

forall (k = 1; k <= N; k++)
  for (l ∈ X_k)
    forelem (i; i ∈ pA_k.field1[l])
    SEQ;

where the loop body accesses, for example, A_k[i].field1. Note that, this data
decomposition guarantees that pA_k only contains subscripts i such that values
A_k[i].field1 are always contained in X_k. Based on this observation, the loop can
be simplified to:
12.3 Iteration Space Expansion

This section introduces the Iteration Space Expansion transformation, which is an important transformation to enable efficient distributed execution of forelem loops. Furthermore, the transformation can turn forelem loops into a form suitable for MapReduce-like processing. Before this transformation is described in detail, a number of related transformations are reviewed.

Previous in Chapter 6 Iteration Space Expansion has been described. Iteration Space Expansion is advantageous in codes that exhibit irregular access patterns that are made regular by iterating the (expanded) iteration space in which the irregular accesses are contained. For example, consider the following loop from a sparse matrix code [95]:

```c
for (i = 0; i < N; i++)
{
    for (q ∈ colIndex(Δ))
    {
        result[i] += M'[i, q] * right[q];
    }
}
```

\[ Δ = \{\text{start}[i], \text{start}[i] + 1, \ldots, \text{start}[i + 1] - 1\} \]
where \textit{colIndex} exhibits an irregular access pattern. The iteration space of the inner loop is expanded to iterate the entire positive integer range:

\begin{verbatim}
for (i = 0; i < N; i++)
    for (q = 0; q < INT_MAX; q++)
        result[i] += M''[i, q] * right[q];
\end{verbatim}

where \( M''[i, q] \) is defined by

\begin{verbatim}
if (q \in \text{colIndex}(\Delta))
    M''[i, q] = M'[i, q];
else
    M''[i, q] = 0;
\end{verbatim}

so that the semantics are preserved, because for subscripts \( i, q \) for which \( M' \) is not defined, \( 0 \) is returned. This code is an intermediate step in an optimization process and enables new optimization opportunities because the two loops are now regular.

For Iteration Space Expansion in the context of \textit{forelem} loops, the following example is considered:

\begin{verbatim}
count = 0;
forelem (i; i \in pA.field[X])
    count++;
tmp = count;
\end{verbatim}

Essentially, the example counts the number of array subscripts for which \textit{field} equals a value \( X \). This can be seen as the computation of an aggregate value, for example as part of a group-by computation where the aggregate is computed for different values \( X \). The example can be rewritten as follows, with the condition made explicit:

\begin{verbatim}
count = 0;
forelem (i; i \in pA)
    if (X == A[i].field)
        count++;
tmp = count;
\end{verbatim}

Let us recap Iteration Space Expansion as described in Chapter 6. The Iteration Space Expansion transformations now consists of three steps. Firstly, the condition on \textit{field} is eliminated, so that the body of the \textit{if} statement (the actual loop body) is executed for all subscripts of \( A \). In fact, the iteration space is expanded from \textit{pA.field}[X] to \textit{pA}. Secondly, the scalar \textit{count} is expanded to a vector, subscripted by \( A[i].field \). Thirdly, any reference to the scalar \textit{count} is rewritten to access the vector, with \( X \) as subscript. The value that is assigned to \textit{tmp} is then equivalent to the value assigned in the original code. This results in:
count[] = 0;
forelem (i; i ∈ pA)
    count[A[i].field]++;
tmp = count[X];

The transformation is generalized as follows, see also Chapter 6. For a loop of the form

forelem (i; i ∈ pA.field[X])
    SEQ;

the following steps are performed:

1. the condition $A[i].field == X$ is removed, which expands the iteration space so that the entire array $A$ is visited,

2. scalar expansion is applied on all variables that are written to in the loop body denoted by $SEQ$ and references to these variables are subscripted with the value tested in the condition, in this case $A[i].field$,

3. all references to the scalar expanded variables after the loop are rewritten to reference subscript $X$ of the scalar expanded variable.

As an additional example of the transformation, consider the following loop which computes the average of a set of values:

count = 0;
sum = 0;
forelem (i; i ∈ pA.field1[X])
    {
        sum += A[i].field2;
        count++;
    }
tmp = sum / count;

When the same transformation steps are carried out, the result is:

count[] = 0;
sum[] = 0;
forelem (i; i ∈ pA)
    {
        count[A[i].field1]++;
    }
tmp = sum[X] / count[X];

A useful application of the Iteration Space Expansion transformation is in loop nests that compute an aggregate function for a series of values. For example:
forelem (i; i ∈ pA.distinct(field1))
{
    count = 0;
    forelem (j; j ∈ pA.field[A[i].field])
        count++;
    R = R ∪ (A[i].field, count)
}

which computes the count aggregate function for all distinct values of field1 in array A. The Iteration Space Expansion transformation is applied to the inner loop, to result in:

forelem (i; i ∈ pA.distinct(field1))
{
    count[] = 0;
    forelem (j; j ∈ pA)
        count[A[j].field]++;
    R = R ∪ (A[i].field, count[A[i].field])
}

The inner loop that computes the count array is now fully independent of the outer loop. Loop Invariant Code Motion is applied to move the inner loop out of the outer loop:

count[] = 0;
forelem (j; j ∈ pA)
    count[A[j].field]++;
forelem (i; i ∈ pA.distinct(field1))
    R = R ∪ (A[i].field, count[A[i].field])

As a result of the preceding transformation, the array A only has to be iterated once to compute all aggregates, at a cost of higher memory usage to store the count array. In fact, the loop resulting from this chain of transformations is similar to a hash aggregation strategy that is used in database systems. Furthermore, the first loop allows for straightforward parallelization:

count[] = 0;
forall (k = 1; k <= N; k++)
    forelem (j; j ∈ pkA)
        count[A[j].field]++;
forelem (i; i ∈ pA.distinct(field1))
    R = R ∪ (A[i].field, count[A[i].field])

12.4 Illustration of the application of transformations

This section illustrates how the distribution of forelem loops over multiple processors described in Section 12.2, the transformations defined in the forelem framework (see for instance Chapter 3), and the Iteration Space Expansion transformation described in the previous section are used together and reinforce each other.
In particular, we will show that two adjacent loops which access the same table and are distributed based on different fields of this table can be transformed so that both loops use the same data distribution and a costly data redistribution is not necessary.

As a starting point, the following two adjacent loops on Table are considered:

```c
forelem (i; i \in pTable)
SEQ;
...
forelem (i; i \in pTable)
SEQ;
```

where the first loop is distributed based on field1 and the second loop on field2:

```c
forall (j = 1; j <= N; j++)
for (k \in X_j)
  forelem (i; i \in pTable.field1[k])
SEQ;
...
forall (j = 1; j <= N; j++)
for (k \in X_j)
  forelem (i; i \in pTable.field2[k])
SEQ;
```

Even if Table.field1 \equiv Table.field2 and the two decompositions are the same, data partitioning conflicts can occur. This is because a partitioning of Table based on field1 is not equal to a partitioning of Table on field2. The fact that the column contents are equal does not imply the column contents are in the same order (the columns are multisets).

To resolve this, either Table is not distributed for the first loop or a redistribution of the table data is performed in between the first and second loop. Evidently, both are suboptimal solutions. However, if the forelem loop bodies compute an aggregate function, then a solution is possible.

For instance, assume SEQ of the first loop consists of incrementing a counter, thereby computing the multiplicity of all values of field1 in Table. An outer loop is required to perform the aggregate function for every distinct field1 value in Table, and the first loop results in:

```c
forelem (i; i \in pTable.distinct(field1))
{
  count = 0
  forelem (j; j \in pTable.field1[Table[i].field1])
    count++;
  R_1 = R_1 \cup (Table[i].field1, count)
}
```

This loop nest is suboptimal as it makes multiple passes through Table in the inner loop. To enable parallelization of this loop nest, Iteration Space Expansion is used, resulting in:
forelem (i; i ∈ pTable.distinct(field1))
{
    // Initialize count to zero for all dimensions
    count[] = ∅
    forelem (j; j ∈ pTable)
    count[Table[j].field1]++;
    \( R_1 = R_1 \cup (Table[i].field1, count[Table[i].field1]) \)
}

Because the computation of count is now independent of i, the loop computing
the count array can be moved out of the enclosing loop:

count[] = ∅
forelem (j; j ∈ pTable)
    count[Table[j].field1]++;
forelem (i; i ∈ pTable.distinct(field1))
    \( R_1 = R_1 \cup (Table[i].field1, count[Table[i].field1]) \)

As a next step, the loop computing the count array is parallelized:

count[] = ∅
forall (k = 1; k <= N; k++)
    for (l ∈ X_k)
        forelem (j; j ∈ pTable.field1[l])
            count[Table[j].field1]++;
forelem (i; i ∈ pTable.distinct(field1))
    \( R_1 = R_1 \cup (Table[i].field1, count[Table[i].field1]) \)

Writes to count are performed to a global array which potentially generates signif-
icient amount of communication. (However, careful analysis will indicate that
the writes to count are in this case controlled by the distribution of \( X \), such that
no two distinct nodes will write to the same subscript of count). The amount of
communication can be reduced by creating a local array \( count_k \) for each processor \( P_k \):

count[] = ∅
forall (k = 1; k <= N; k++)
{
    count_k = ∅
    for (l ∈ X_k)
        forelem (j; j ∈ pTable.field1[l])
            count_k[Table[j].field1]++;
}
forelem (i; i ∈ pTable.distinct(field1))
{
    count[Table[i].field1] = \( \sum_{k=1}^N count_k[Table[i].field1] \)
    \( R_1 = R_1 \cup (Table[i].field1, count[Table[i].field1]) \)
An alternative parallelization approach is to parallelize both loops and not only the loop computing the count array. As a consequence of parallelizing the loop creating the result table, the processors should create partial result tables \( \mathcal{R}_{1,k} \) which are later combined. Both loops are parallelized with \( X = \text{Table.field1} \), which results in:

```plaintext
forall (k = 1; k <= N; k++)
  for (l \in X_k)
  {
    forelem (i; i \in \text{pTable.field1}[l])
    count_{k}[\text{Table}[i].field1]++
    forelem (i; i \in \text{pTable}.distinct(field1))
    \mathcal{R}_{1,k} = \mathcal{R}_{1,k} \cup (\text{Table}[i].field1, \text{count}_{k}[\text{Table}[i].field1])
  }

sum[] = 0
forall (k = 1; k <= N; k++)
  forelem (i; i \in \text{p}\mathcal{R}_{1,k})
  sum[\mathcal{R}_{1,k}[i].field1] += \mathcal{R}_{1,k}[i].count
  forelem (i; i \in \text{pTable}.distinct(field1))
  \mathcal{R}_1 = \mathcal{R}_1 \cup (\text{Table}[i].field1, \text{sum[Table}[i].field1])
```

Now, we return to the initial example. We consider two of the above loops, where the former is parallelized with \( X = \text{Table.field1} \) and the latter with \( X = \text{Table.field2} \). Then the first two `forall` loops are:

```plaintext
forall (k = 1; k <= N; k++)
  for (l \in X_k)
  {
    forelem (i; i \in \text{pTable.field1}[l])
    count_{1,k}[\text{Table}[i].field1]++
    forelem (i; i \in \text{pTable}.distinct(field1))
    \mathcal{R}_{1,k} = \mathcal{R}_{1,k} \cup (\text{Table}[i].field1, \text{count}_{1,k}[\text{Table}[i].field1])
  }

...```

```plaintext
forall (k = 1; k <= N; k++)
  for (l \in X_k)
  {
    forelem (i; i \in \text{pTable.field2}[l])
    count_{2,k}[\text{Table}[i].field2]++
    forelem (i; i \in \text{pTable}.distinct(field2))
    \mathcal{R}_{2,k} = \mathcal{R}_{2,k} \cup (\text{Table}[i].field2, \text{count}_{2,k}[\text{Table}[i].field2])
  }
```

As has been indicated at the beginning of this section, data partitioning conflicts will occur for these two loops. These could be solved by performing an expensive data redistribution in between the execution of these two loops. However, in this case a better solution is to exploit the possibility to reorder the loops such that the two parallelized loops computing the `count` aggregate are consecutive to one another. This is possible because these loops do not have a dependency on the other
loops (the second forall loops) in the code fragment. The two outermost loops iterate the same bounds, allowing application of the Loop Fusion transformation:

```plaintext
forall (k = 1; k <= N; k++)
for (l ∈ X_k)
{
    forelem (i; i ∈ pTable.field1[l])
    count1,k[Table[i].field1]++
    forelem (i; i ∈ pTable.distinct(field1))
    R1,k = R1,k ∪ (Table[i].field1, count1,k[Table[i].field1])
    forelem (i; i ∈ pTable.field2[l])
    count2,k[Table[i].field2]++
    forelem (i; i ∈ pTable.distinct(field2)))
    R2,k = R2,k ∪ (Table[i].field2, count2,k[Table[i].field2])
}
```

In the case that Table.field1 ≡ Table.field2, another series of statement re-ordering and Loop Fusion is possible in the loop body resulting in:

```plaintext
forall (k = 1; k <= N; k++)
for (l ∈ X_k)
{
    forelem (i; i ∈ pTable.field1[l])
    {
        count1,k[Table[i].field1]++
        count2,k[Table[i].field2]++
    }
    forelem (i; i ∈ pTable.distinct(field1))
    R1,k = R1,k ∪ (Table[i].field1, count1,k[Table[i].field1])
    forelem (i; i ∈ pTable.distinct(field2))
    R2,k = R2,k ∪ (Table[i].field2, count2,k[Table[i].field2])
}
```

Because the two counting loops use the same partitioning of X, it is possible to fuse these two loops. In other words, the loops use the same data distribution and no data redistribution is necessary in between loops. This technique can be extended to other combinations of loops, such as for example:

```plaintext
forelem (i; i ∈ pTable)
R = R ∪ (...) 
...
forelem (i; i ∈ pR)
SEQ;
```

The second loop consumes tuples produced by the first loop. If the second loop does not have any restricting dependencies, the body of the second loop can be moved to the position in the first loop where the tuples are produced. As a result, also in this case both loops make use of the same data distribution of Table.
Although the interaction of the different transformations is rather powerful, it should be noted that we have only considered one particular case of two consecutive *forelem* loops. In general, database applications are not that simple and consist of many queries, embedded or not embedded in application code, so, the complexity of these interactions will grow exponentially. Although not addressed in this thesis, it is important to reckon that strategies will have to be developed to keep the optimization process manageable.

### 12.5 Application on Big Data Programs

In this section, it will be illustrated how the described *forelem* framework is used to optimize Big Data applications. The two examples from the original MapReduce [30] paper are considered which process data that is typically acquired from the use of the World Wide Web: web server page request logs and a database of links between web pages. We show that, using the *forelem* framework and starting with a SQL representation of the problem, a MapReduce-like program can be automatically derived and that transformations can be applied as usual on the *forelem* representation of the problem. Secondly, to show the importance of a good data layout, we explore the performance of different codes generated for these examples, using different data layouts, and compare this performance to the implementation of the examples in Hadoop.

#### 12.5.1 URL Access Count

The first example concerns URL access count. Consider logs of web page requests, which are mapped to tuples \((url, 1)\). The reduction operator is described in the paper as mapping \((url, list(values))\) to \((url, total\_count)\). Considering a table *access*, with a single column containing the URLs, this computation can be described as the following SQL query:

\[
\text{SELECT url, COUNT(url) FROM access GROUP BY url}
\]

The *forelem* framework will generate the following loop nests from this query:

**forelem** \((i; i \in pAccess.distinct(url))\)
\[
\mathcal{G} = \mathcal{G} \cup (\text{Access}[i].url)
\]

**forelem** \((i; i \in p\mathcal{G})\)
\[
\begin{align*}
\text{count} & = 0; \\
\text{forelem} \ (j; j \in pAccess.url[\mathcal{G}[i].url]) & \\
\text{count} & = \text{count} + 1; \\
\mathcal{R} & = \mathcal{R} \cup (\mathcal{G}[i].url, \text{count})
\end{align*}
\]

The inner loop of the *forelem* loop iterating \(p\mathcal{G}\) suits the application of the Iteration Space Expansion transformation:
forelem (i; i ∈ pAccess.distinct(url))
  \( \mathcal{I} = \mathcal{I} \cup (\text{Access}[i].url) \)
forelem (i; i ∈ p\( \mathcal{I} \))
  {
    count[] = 0;
    forelem (j; j ∈ pAccess)
      count[Access[j].url]++;
    \( \mathcal{R} = \mathcal{R} \cup (\mathcal{I}[i].url, \text{count}[\mathcal{I}[i].url]) \)
  }

As the inner loop has been made fully independent of the enclosing loop, the loop can be moved outwards:

forelem (i; i ∈ pAccess.distinct(url))
  \( \mathcal{I} = \mathcal{I} \cup (\text{Access}[i].url) \)
count[] = 0;
forelem (j; j ∈ pAccess)
  count[Access[j].url]++;
forelem (i; i ∈ p\( \mathcal{I} \))
  \( \mathcal{R} = \mathcal{R} \cup (\mathcal{I}[i].url, \text{count}[\mathcal{I}[i].url]) \)

The transformation sequence so far has turned the multiple irregular accesses to the Access array into a single regular iteration. As a next step, the first and third loops are merged using a transformation called Table Propagation, which will propagate the loop creating table \( \mathcal{I} \) to the loop accessing \( \mathcal{I} \), effectively eliminating the streaming of a table between two loops:

count[] = 0;
forelem (j; j ∈ pAccess)
  count[Access[j].url]++;
forelem (i; i ∈ pAccess.distinct(url))
  \( \mathcal{R} = \mathcal{R} \cup (\text{Access}[i].url, \text{count}[\text{Access}[i].url]) \)

The first loop is a good candidate for parallelization. The index set pAccess is divided over \( N \) processors (direct loop scheduling) such that the loop can be executed in parallel:

count[] = 0;
forall (k = 1; k ≤ N; k++)
  forelem (j; j ∈ p\( \text{k} \)Access)
    count[Access[j].url]++;
forelem (i; i ∈ pAccess.distinct(url))
  \( \mathcal{R} = \mathcal{R} \cup (\text{Access}[i].url, \text{count}[\text{Access}[i].url]) \)

A problem with this loop nest is that all processors write to the shared count array to store the results, which will be a big bottleneck in the code that will be generated from this intermediate representation. To alleviate this, every processor is given a local count array, from which a single value is reduced in the second loop:
count_{1...N}[] = 0;
forall (k = 1; k <= N; k++)
  forelem (j; j ∈ p_kAccess)
    count_k[Access[jj].url]++;
forelem (i; i ∈ pAccess.distinct(url))
  \mathcal{R} = \mathcal{R} \cup (Access[i].url, \sum_{k=1}^{N} count_k[Access[i].url])

Note that this code fragment bears similarity to a MapReduce program. In fact, the first loop maps every row of access to an accumulation of the Access[i].url subscript of the count array. This could be represented as a tuple (url, 1). The second loop iterates over all keys, which are all distinct URLs in access and retrieves the result of an aggregate function, in this case count.

In general, two adjacent forelem loops where the former loop stores values in an array subscripted by a field of the array being iterated, and the latter loop accesses elements of this array, can be written as a MapReduce program. The map function iterates the table that is iterated by the former loop. This table is fragmented by a MapReduce framework, so that each instance of the map function processes a table fragment. This corresponds with a data distribution for the above code fragment where each processor has the rows that are referenced by the index set fragment stored locally. Instead of writing to a global array, emitIntermediate is called. For the above example, tuples (Access[i].url, 1) are generated, where the 1 is a dummy value, because it is not used.

The example code increments the value stored in the count array for every occurrence of a value Access[i].url. In the MapReduce program, a pair will be generated for every Access[i].url. So, the reduction function has to increment a counter for every occurrence of the same value Access[i].url. Because a MapReduce framework will collect all pairs for a unique key, the reduction function simply needs to count all values for every unique key. If the above example is written in MapReduce pseudocode similar to that used in [30], the program would be:

map(key, value):
  # Assume value represents the content of the
  # access table
  access = value
  for a in access:
    emitIntermediate(a.url, 1)

reduce(key, values):
  count = 0
  for v in values:
    count++
  emit(key, count)

12.5.2 Reverse Web-Link Graph

As a second example from the MapReduce paper we consider the Reverse Web-Link Graph. For each link from a source to a target page, a pair (target, source)
is emitted. The original example reduces to a pair \((target, list(source))\), which we will modify to reduce to a pair \((target, source.count)\). To write a SQL query for this program, consider a table \(links\) that contains tuples \((source, target)\), which has been previously filled, for example by parsing webpages \(source\) and extracting all links to target pages. The following two queries are defined:

\[
\text{CREATE VIEW target_links AS}
\]
\[
\text{SELECT DISTINCT target FROM links;}
\]
\[
\text{SELECT T.target,}
\]
\[
\left(\text{SELECT COUNT(*) FROM links L}
\text{WHERE L.target=T.target}\right)
\]
\[
\text{FROM target_links T;}
\]

which compute the number of incoming links to each registered target page. Expression of these queries in the \textit{forelem} framework results in the following loops:

\[
\text{forelem (i; i \in pLinks.distinct(target))}
\]
\[
\mathcal{T} = \mathcal{T} \cup (\text{links}[i].target)
\]
\[
\text{forelem (i; i \in p\mathcal{T})}
\]
\[
\{
\text{count = 0;}
\text{forelem (j; j \in pLinks.target[\text{links}[i].target])}
\text{count++;}
\mathcal{R} = \mathcal{R} \cup (\text{links}[i].target, \text{count})
\}
\]

Using Table Propagation (see Section 6.2.3, this can be turned into a single loop nest:

\[
\text{forelem (i; i \in pLinks.distinct(target))}
\]
\[
\{
\text{count = 0;}
\text{forelem (j; j \in pLinks.target[\text{links}[i].target])}
\text{count++;}
\mathcal{R} = \mathcal{R} \cup (\text{links}[i].target, \text{count})
\}
\]

Let us consider a different transformation chain for this example. The outer loop iterates all distinct values of \(target\). In fact, the value range of \(\text{Links.target}\) is iterated. Let \(X = \text{Links.target}\) and parallelize the loop using indirect loop scheduling:

\[
\text{forall (k = 1; k <= N; k++)}
\]
\[
\{
\text{for (l \in X_k)}
\}
\]
\[
\text{count = 0;}
\text{forelem (j; j \in pLinks.target[l])}
\text{count++;}
\]
\[ R = R \cup (1, \text{count}) \]

Subsequently, Iteration Space Expansion is applied on the inner loop and the loop is moved outwards one level:

```c
forall (k = 1; k <= N; k++)
{
  count[] = \emptyset;
  forelem (j; j \in pLinks)
    count[links[j].target]++;
  for (l \in X_k)
    R = R \cup (l, \text{count}[l])
}
```

In the current loop, every processor will compute its own copy of the `count` array. Also, the processors will contend for access to the result table `R`. One possibility is to give every processor a private copy of `R` and merge the copies to a final result table in the master node:

```c
forall (k = 1; k <= N; k++)
{
  count[] = \emptyset;
  forelem (j; j \in pLinks)
    count[links[j].target]++;
  for (l \in X_k)
    R_k = R_k \cup (l, \text{count}[l])
}
R = \bigcup_{k=1}^{N} R_k
```

Another possibility is to move the loop computing the array further outwards:

```c
count[] = \emptyset;
forelem (j; j \in pLinks)
  count[links[j].target]++;
forall (k = 1; k <= N; k++)
  for (l \in X_k)
    R = R \cup (l, \text{count}[l])

And to undo the parallelization of the second loop:
```

```c
count[] = \emptyset;
forelem (j; j \in pLinks)
  count[links[j].target]++;
forelem (i; i \in pLinks.distinct(target))
  R = R \cup (links[i].target, \text{count}[links[i].target])
```

Instead of parallelizing the second loop, the first loop can be selected for parallelization. This will result in loops similar in structure to the first example (see Section 12.5.1).
12.5.3 Initial Performance Comparison

A number of initial experiments have been conducted with Hadoop and forelem-generated implementations of the two described examples. Different implementations were generated with the forelem framework, using different data layouts. The experiments have been performed on the DAS-4 cluster at Leiden University [48]. The cluster nodes each contain 8 processing cores, 48GB of main memory and 10 TB of local storage in a software RAID0 configuration. The Hadoop experiments were performed on a Hadoop cluster of 7 data nodes and one master node running the task tracker. The forelem implementation is a C code generated using the forelem framework, which uses MPI and OpenMP message exchange and local parallelization. This implementation is also run on 7 nodes and one separate master node.

The Access Count example has been run on a generated data file of 320GB, which is a comma separated file containing URL, data, server name that processed the request and a status code. The file has been stored onto the HDFS for processing with Hadoop and was evenly distributed over nodes according to a static, direct, loop schedule for processing by the forelem implementation. The Reverse Link count example has been run on a comma separated file containing source and target URL pairs. This file had a size of 177GB.

The results of these experiments are visualized in Figures 12.1 and 12.2. The numbers shown are averages of 4 runs, the variance between the experiments is negligible. The experiments show that the forelem implementations realize a performance improvement of a factor 3 when the same input data is used as is used by Hadoop, and up to a factor 120 if the input data is available with an optimized layout. It should be noted that if it is possible to reformat the data, large performance improvements can be achieved.
Different versions of the forelem implementation have been generated. The first reads the text file in the same format as the Hadoop implementation. The code that has been generated from the forelem loop does not iterate through a pre-formatted array, but instead iterates the lines in the text file, which are split into the separate fields.

In the other experiments the use of a binary file that contains the pre-formatted array has been studied. This array is mapped into memory and processed by the code generated from the forelem intermediate representation. The experiments show that the use of such a format is not beneficial if the data file contains strings, due to the padding required in the binary format. As a result, the binary files are considerably larger than the text files and time taken by the I/O subsystem to read this file does not weigh up to the savings in parsing. When parallelism is reduced to a single core per node (i.e. to minimize the amount of disk seeks triggered by multiple processes reading different data from the disk) the performance does not improve. Instead, the CPU was kept busy, indicating that the disk read speed is not a problem when only a single thread per node is used. It is likely that there is a sweet spot where the number of threads is in balance with the throughput from the I/O system.

The forelem framework is capable of automatically reformatting the data layout of a program. As an example, the strings (URLs and hosts) in the arrays have been replaced with integer keys. These integer keys are used to subscript another array, which contains the string value for each key. In fact, the data model has been made relational. This significantly improves the performance, as indicated by the “integer keyed” experiments, which implies that it is worthwhile to consider such data reformatting if this is feasible in the context of the problem, for example when the data has not yet been collected in a specific format. A final experiment has been done by removing unused structure fields and column-wise storage of
the data. These data relayout operations can also be done automatically by the forelem framework. A performance increase is not observed after performing this relayout, possibly because it does not weigh up to the initial start up cost of the MPI and OpenMP frameworks.

12.6 Conclusions

This chapter described an extension of the forelem to express distributed execution of forelem loops. These extensions enable the forelem framework to exert control over the data distribution and decomposition across multiple compute nodes in addition to the control of (local) data layout as was discussed in Chapter 9.

In the context of distributed forelem codes, the Iteration Space Expansion transformation plays an important role. Through the use of this transformation and the other transformations present in the forelem framework, changes are not only made to the loop structure of the program but also to the data layout by reformatting this layout. Also, it has been described how these extensions can be used to translate a problem expressed in SQL to a parallelized forelem representation of the problem, from which a MapReduce-like program can be deduced.

The viability to use the forelem framework to optimize Big Data applications has been illustrated using two example MapReduce problems. These problems have been expressed in the forelem intermediate representation and were subsequently optimized. From this forelem intermediate representation different codes have been generated, using different data layouts. The performance of these different codes have been compared to a Hadoop implementation of the same problem. From these initial experiments follows that data layout plays an important role. When the same data file is used, performance improvements were obtained of at least a factor 3. If it is possible to reformat the data, the implementations generated using the forelem framework with reformatted data show performance improvements up to a factor 120.