Compiler Optimizations for Java Aglets in Distributed Data Intensive Applications

Abhishek Singh , Santosh Pande
College of Computing
801 Atlantic Drive
Georgia Institute of Technology
Atlanta, GA 30332
{abhi,santosh}@cc.gatech.edu

Abstract

Java Aglets offer a powerful paradigm for writing applications for distributed systems involving data mining, web crawling etc. A Java aglet allows migration of mobile code on distributed platforms along with the data and the state of evaluation. This facilitates partial evaluation of a program at a given computing node.

Code migration in the light of distributed data intensive computing poses interesting compiling issues. In this work, we first define a small extension to the aglet model to allow data distribution. In our aglet program, data are distributed over the network using annotations (this is similar to HPF where the programmer specifies data distributions through annotations). We analyze the program using annotations and data sizes and use the ‘owner computes’ rule to determine where a given computation should take place.

Our compiler infrastructure called Compiler Scheduler (CS) then schedules the aglet through the network. We propose two strategies to optimize the aglet schedule. The first strategy called Take All Live Data (TALD) attempts to carry all the live definitions of variables from a given node when visited. The second strategy Take Only Needed Data (TOND) attempts to carry only those definitions whose uses are in the destination node. The goal of the first strategy is to minimize the number of migrations. Migrations are expensive because the serialization of data encountered in each migration can be in the order of milliseconds. The second strategy aims to minimize bandwidth consumption during a migration. This could significantly reduce the communication overhead due to minimal amount of data carried during each migration. We have developed our (compiler scheduler) infrastructure by implementing both the strategies in the Jikes compiler from IBM. We have evaluated it on a distributed database application and show benefits of both the strategies on large and small databases. We have also evaluated our strategies against typical distributed operations on data such as Gather, Fusion, Consistency Check and MergeSort and compare our schedules against randomized ones. The results show that strategies generated by our compiler infrastructure out preform random strategies.

1 Introduction

With the advent of the Internet, applications that access distributed data have become prominent. With the emergence of Java, this has become simple, due to the portability of Java. The most popular method used by distributed applications was the Client-Server paradigm, where the application establishes a connection with the data server and obtains the data. CORBA and Java implement this through the RMI mechanism. This requires that the connection be persistent as long as there is a data transfer. However, with the emergence of highly distributed data sources, the class of applications which need truly symmetric distributed processing (not limited just to the Client-Server model) are becoming increasingly important. Examples of these applications include distributed banking transactions, e-commerce database transactions and military sensor data fusion codes.

The idea of using mobile code for distributed computing has evolved recently. Various design paradigms for a mobile code have been elaborated by A. Carzingsa et. al [1]. The Mobile Agent paradigm [6] facilitates the movement of code and state of execution from one machine to another during the course of evaluation. Mobile agents are very useful for the distributed applications that operate on a enormous amount of data. They save the network bandwidth and communication costs by moving code to the location where the data are available rather than transferring the huge amount of data [7]. Interesting issues are encountered when determining the schedule of a mobile agent through a distributed system. The performance of a schedule can be enhanced by carefully optimizing the number of migrations of a mobile agent through the network and the amount of data carried during each migration through compiler analysis which is the theme of this paper.

1.1 Mobile Agents

Mobile agents are objects which consist of the code (an algorithm), data and state (execution thread along with the stack) and can move from machine to machine under their own control. Mobile agents can be classified as Strong and Weak agents depending on the kind of migration they provide. Strong Agents are those that take care of transporting the execution stack themselves and continue the computation from their last state. Weak Agents are those that rely on the programmatic steps to continue the evaluation. Normally, all Java based mobile agents are weak with respect to migration as Java doesn’t support thread migration.
Many researchers are investigating the mobile agent systems. The performance of Mobile agents depend on the kind of applications they are employed in. The performance evaluation of the mobile agent model against the client/server model has been explained in [10] using RMI and Aglets Mobile Agent Platform [9] [17]. It also identifies the applications that can perform well using mobile agents, despite the cost of the mechanisms provided by Java for implementing Mobile agents. Sträßer [11] presents a performance model for interaction between agents. Many mobile agents like Telescript, Hive, Aglets, Voyager, Ajanta, D’Agents, Mole [14] have been developed during the last few years. Most of them are implemented in Java.

1.2 Compiler Optimizations

This work attempts to perform optimizations for Java mobile agents code on distributed data intensive applications. The compiler takes as input an annotated Java program, that accesses distributed data.

The annotations describe the distribution of (both persistent and non-persistent) program variables over the network nodes. The compiler, based on the annotations, identifies a legal schedule for the computation using dependence information. The schedule is optimized to minimize the total volume of data carried by the aglet. Further, the generated schedule is optimized to minimize either the maximum amount of data carried during a migration (the goal is to reduce the latency due to a limited bandwidth for data intensive application) or the number of migrations themselves (Serialization which makes a migration costly).

2 Motivation and Outline of Approach

```java
public class foo{
  int[] A;///\(A,PE1.gatech.edu,1000\);
  int[] B;///\(B,PE1.gatech.edu,1000\);
  int[] C;///\(C,PE1.gatech.edu,1000\);
  int[] D;///\(D,PE2.gatech.edu,1000\);
  int[] E;///\(E,PE3.gatech.edu,1000\);

  public void fun1(){
    L1 : for(int i=0; i<1000;i++)
      A[i] = B[i]+C[i];
  }

  public void fun2(){
    L2 : for(int i=0; i<1000;i++)
      D[i] = A[i]+B[i];
  }

  public void fun3(){
    L3 : for(int i=0; i<1000;i++)
      E[i] = A[i]+C[i];
  }
}
```

Consider the above example code which accesses the data distributed across three processing nodes (PEs) in a network. The annotation `/\(A,PE1.gatech.edu,1000\)` indicates that array `A[]` resides at computation node `PE1.gatech.edu` and has a maximum size of 1000. In our work, we currently employ the `owner computes` rule, which stipulates that we perform the computation at a processing node which `owns` the `l-value` of a variable (this is a typical rule used in parallel and distributed computing and HPF and many other languages use this). Thus, we will compute `L1` at `PE1` (since node `PE1` `owns` the `l-value` for `L1` which is `array A[]`). Similarly `L2` is computed at `PE2` and `L3` at `PE3`. Figure 1 shows the data dependence graph of the code. The legal schedules are execution of `L1` followed by `L2` followed by `L3` or execution of `L1` followed by `L3` followed by `L2`. Consider a legal schedule `L1 -> L2 -> L3`; in order to execute it, the Java Mobile Agent (aglet) will first go to `PE1`, execute `L1`, migrate to `PE2`, execute `L2` and will finally migrate to `PE3` and execute `L3`. At each node it will have to fetch variables it needs from respective computation nodes causing further migrations. The number of migrations and amount of data carried during each migration determine the overheads on mobile code execution and thus, the relative tradeoff between the two determines the performance. This is where compiler optimizations in scheduling the aglet and deciding which variables to carry with it and which ones to fetch (when needed) can make a difference as shown below.

When the aglet executes one of the above schedules, it migrates from one PE to another as per schedule as illustrated above. During a migration, it can choose to carry only the data it needs at destination node (let’s call it `TakeOnlyNeededData()` policy). Alternatively it can choose to carry all the data which is needed not only at the destination node but the one which is live and needed at all the nodes which follow the destination node in a given schedule (let’s call it `TakeAllLiveData()` policy). Thus, in the above schedule by the `TakeOnlyNeededData()` policy, the aglet will carry `A[]` and `B[]` needed at `PE2` during its migration from `PE1`. During its migration from `PE2` to `PE3`, it will carry `A[]`, `B[]` and `C[]` as it is needed at `PE3`. However, this would require `PE1` to get it. Thus, the total number of migrations under this policy will be four (`PE1` to `PE2` to `PE2` to `PE2` to `PE3`). However, one can see that the amount of data carried during each migration is minimal (maximum size is 2000 units). This will allow the aglet to stay thin and move quickly during each migration.

Under the policy it `TakeAllLiveData()`, on the other hand, the aglet will carry all three `A[]`, `B[]`, `C[]` when migrating from `PE1` to `PE2`. This is due to the fact that although there is no use of `C[]` at `PE2`, there is a use of `C[]` during the execution of `L3` on `PE3` which follows `PE2` in the schedule. When the aglet migrates from `PE2` to `PE3`, it will only carry `A[]`, `C[]` to it. In this case, the aglet has all the data it needs at `PE3` and thus, it does not have to migrate to get `C[]` from `PE1` unlike previous policy. Thus, the number of migrations here are three instead of four in the previous case. The maximum amount of data carried during a migration, however is larger (3000 units from `PE1` to `PE2`) than previous policy and this might cause an aglet to migrate slowly.

Depending upon the tradeoff between overhead of serialization and communication slowdown due to smaller bandwidth during migration, one policy might do better than other. Our results section shows that for large databases `TakeOnlyNeededData` performs better whereas for smaller ones, `TakeAllLiveData` does better on a dedicated LAN.

3 Framework for Optimizing Mobile Agents

The main contributions of this work are devising an efficient algorithm for scheduling an aglet through the network and determining the sets of variables that should be carried when
an aglet migrates.

Our algorithm generates an efficient schedule for the aglet using its data dependence graph (DDG). We traverse the DDG in topological order and schedule each node of DDG sorted in the increasing order of their data requirement. This approach is motivated to reduce the total amount of data carried through the network. Its benefit in the case of TALD policy is shown in the illustrative example in section 3.3.

The two policies (TALD and TOND) for data transfer optimizations are described below. The compiler chooses either strategy based upon the size of the application.

**TakeAllLiveData (TALD)** In this strategy the mobile agent carries all the live variables along with the computed result when it migrates from a node to another. These variables need not have immediate uses in the destination processing node. This strategy is selected by the compiler when a minimum number of migrations is desired. As can be seen, a processing node with a live definition of a variable is visited only once to acquire and carry this definition. The drawback of this strategy is that it forces the aglet to carry the data which may not have immediate use and thus, the aglet becomes ‘fat’ during the migrations. This can be quite severe for large database and can cause slowdown.

**TakeOnlyNeededData (TOND)** In this strategy a mobile agent will carry only the variables which have an immediate use in the destination processing node in a schedule. The variables which may be live (that is they have uses somewhere down the road in a given schedule) but which do not have immediate uses in destination node are not carried. This strategy allows the aglet to remain ‘thin’ during each migration (and is thus good for applications which are based on large amounts of data). It however, has a larger overhead due to higher number of migrations (and thus, serializations) compared to previous strategy.

We now describe the compiler algorithms to generate an aglet schedule and to generate the code for migrations and for carrying variables between migrations.

### 3.1 Terms and Definitions

We first define some terms and definitions used in the analysis.

In conjunction with the algorithm, we define the following terms.

- **Node Set (i)**: This set contains the set of DDG nodes which are at same level number i after topological sort.
- **Used Variable Set (S)** $UVS(S)$: This set contains the set of variables which are being used at a DDG node $S$.
- **SG (i)**: Generated schedule of DDG nodes at level i.
- **First (SG(i))**: First Node scheduled in SG(i).
- **Last (SG(i))**: Last Node scheduled in SG(i).
- **VarSet (SG (i), k)**: Set of all variables to be carried until destination node k as per generated schedule SG(i). We compute and use this set for the TALD policy. This set contains all the variables which are live and are needed in nodes k and beyond as per schedule SG(i) (which is a union of Used Variable Sets of these nodes). The formal computation of this set in TALD is shown in the algorithm below.
- ** CarryVarSet (S1, S2)**: Set of all variables which need to be carried from node S1 to node S2 such that they have a use in node S2 and they were either defined in node S1 or had a use in node S1. CarryVarSet(S1, S2) = $UVS(S2) \cap (UFS(S1) \cup def(S1))$, where, def(S1) is the set of all variables defined in node S1. We use this set to determine variables which should be carried from one node to the next as per TOND policy.
- **FetchSet (S)**: This is the set of variables which must be fetched from other nodes under the given schedule since the aglet did not bring them along during its migration from previous node. Thus, FetchSet(S) = $UFS(S)$ - CarrySet(S1, S), where node S1 is scheduled immediately before S. Again, this is the set of variables which must be fetched from other nodes and we use it during the code generation of TOND policy.

### 3.2 Algorithms

**Input**: Data Dependence Graph (DDG), the Data Distribution Table (DDT) and the Strategy (TALD or TOND).

**Output**: Optimized Schedule and Code for the Mobile Agents

**Procedure FindOptimizedSchedule**

Begin

1. Construct $Node\ set(i)$ by traversing the DDG in Topological order.
2. For each level $i \in Node\ Set(i)$ do the following
   a. Construct the $SG(i) = Gen\ Schedule(i)$
   b. If selected strategy is TALD then,
i. For all nodes $k \in SG(i)$
   A. Compute $VarSet(SG(i), k)$.
   
   $$VarSet(SG(i), k) = \cup_{j \in UVS(k)}$$
   
   , where $j$ is the set of nodes scheduled after $k$ in schedule $SG(i)$.
   /* Find out the variables needed at all the nodes which follow node $k$ in the current
   schedule $SG(i)$ — those must be carried per TALD */

   B. Generate code to carry variables from $Last(SG(i+1))$ to $First(SG(i))$
   using $VarSet(SG(i), 1)$

   C. For all nodes $j \in SG(i)$, $\{j > 2\}$
   generate code to carry variables from $(j-1)$ to $j$ using $VarSet(SG(i), j)$

   (c) If selected strategy is TOND then,

   i. Compute $CarryVarSet( Last(SG(i+1)), First(SG(i)))$.

   ii. Generate code for carrying this set of variables.

   iii. For all variables $\in FetchSet(First(SG(i)))$ determine their processor node as using data
   distribution table and generate fetch code.

   iv. Repeat all of the above steps for each of the nodes in the order scheduled in $SG(i)$.

End

3.3 Illustrative Example

In this subsection we will illustrate gains due to our proposed optimizations the following code. The Data Dependence Graph (DDG) is given in figure 2. Refer to table 1 which shows the data distribution at each processing node (PE).

```java
public class foo {
    int[] a; //b(a,PE1.gatech.edu,4000);
    int[] b; //b(b,PE2.gatech.edu,4000);
    int[] c; //b(c,PE3.gatech.edu,4000);
    int[] d; //b(d,PE4.gatech.edu,4000);
    int[] e; //b(e,PE5.gatech.edu,4000);
    int[] f; //b(f,PE6.gatech.edu,4000);
    int[] g; //b(g,PE7.gatech.edu,4000);
    int[] k; //b(k,PE8.gatech.edu,4000);
    public void fun1() {
        a = new int[4000];
        S1 : for(int i=0;i<4000;i++)
            a[i] = 10;
    }
    public void fun2() {
        b = new int[1000];
    }
}
```

End
Table 1: Data Distribution Table of Illustrative Example

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Node</strong></td>
<td>PE1</td>
<td>PE2</td>
<td>PE5</td>
<td>PE3</td>
<td>PE4</td>
<td>PE6</td>
<td>PE7</td>
<td>PE8</td>
</tr>
<tr>
<td><strong>Defined Variables</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>f</td>
<td>g</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td><strong>Used Variables</strong></td>
<td>a</td>
<td>b</td>
<td>a,b</td>
<td>a,b,c,e</td>
<td>b,d</td>
<td>b,d</td>
<td>g,f</td>
<td></td>
</tr>
<tr>
<td><strong>Available Variables for TOND</strong></td>
<td>a</td>
<td>a,b</td>
<td>b,c,b</td>
<td>d,a,c,b,e</td>
<td>b,d,f</td>
<td>b,d,g</td>
<td>f,k</td>
<td></td>
</tr>
</tbody>
</table>

S2: for(int i=0; i < 1000; i++)
    b[i] = a[i+4]+200;
}

public void fun3()
{
    c = new int[4000];
    S3: for(int i = 0, j=0 ,i<1000;i++,j++)
        c[j] = a[j]+b[i]+ 300;
}

public void fun4()
{
    e = new int[2000];
    S5 : for(i=0;i<1000; i++)
        e[i] = b[i];
        e[i+1000] = 200+b[i];
}

public void fun5()
{
    d = new int[4000];
    S4:for (i=0;i<2000;i++)
        d[i] = a[i]+c[i]+b[i]+e[i];
    for (i=2000;i<4000;i++)
        d[i] = a[i] *c[i];
}

public void fun6()
{
    g = new int[4000];
    S7 :for(int i=0,int j=0;i<1000;i++,j++)
        g[j] = b[i]+d[j];
}

public void fun7()
{
    f = new int[2000];
    S6:for (i=0,int j=0;i<1000;i++,j++)
        f[j] = b[i]+d[j+2];
}

public void fun8()
{
    k = new int[2000];
    S7 : for(int i = 0; i<2000; i++)
        k[i] = g[i]+2]+f[i];
}

After visiting nodes in a topological order, some of the possible legal schedules formed are:

(S1,S2,S3,S5,S4,S6,S7,S8)
(S1,S2,S3,S5,S4,S7,S8)
(S1,S2,S5,S3,S4,S6,S7,S8)
(S1,S2,S5,S3,S4,S6,S7,S8)

We determine a good legal schedule by executing the scheduling algorithm given in last section. According to our algorithm, node S1 at level number 0 is visited first. Node S2 which is at level number 1 is visited next. At level number 2 there are two nodes S3 and S8. Since the data requirement at S5 (2000 bytes) is less than S3 (4000 bytes) S5 is scheduled before S3. Then node S4 at level 3 is scheduled. At level 4 node S6 is scheduled before node S7 using a similar criterion. Finally node number S8 is visited. So the generated schedule according to our strategy is S1,S2,S5,S3,S4,S6,S7,S8.

It may be noted that this schedule is generated with a motivation to make TakeAllLiveData() (TALD) strategy more viable as shown below. The total amount of data movement in the network resulting from performing TALD on the above schedule is 

\[ 4000 + (4000 + 1000 + 2000) + (4000 + 1000 + 2000 + 4000) +
   (4000 + 1000 + 2000 + 4000 + 4000) +
   (4000 + 1000 + 2000 + 4000 + 2000 + 4000) +
   (4000 + 1000 + 2000 + 4000 + 4000 + 2000 + 4000) = 76,000 \text{ bytes} \]

However, for other schedules the amounts of data moved in network are \( 84,000 \text{ bytes} \), \( 84,000 \text{ bytes} \) and \( 82000 \text{ bytes} \). Hence our proposed strategy guarantees minimum data movement among all the possible legal schedules for TALD.

It may be noted that for this example, the maximum amount of data movement takes place in the last migration. The amount of data moved in the last migration is 19,000 bytes.

According to our second strategy TakeOnlyNeededData() TOND, we minimize the amount of data carried at each migration at the cost of increase in number of migrations. In TOND, Node S1 at level 0 is visited first and variable a is carried. Node S2 is then visited. Here variable b is computed. Since node S5 is scheduled next and there is no use of variable a in S5, the variable a is dropped and not carried to node S5. After visiting S5, node S3 is visited. Computation of variable c requires variable a thus PE1.gatech.edu is revisited to fetch variable a; however, it does not need variable e generated in S5 and is thus not carried to S3. Before revisiting PE1 to fetch variable a aglet visits PE3 drops variable b and then performs revisit to fetch a for computation of c. After executing node S3 at PE3 aglet has a, b, c. After computing S3 at node PE3 we proceed to level number 3. Since at level 3 node S4 needs e in addition to a, b, c and thus, aglet fetches it from PE5.gatech.edu. Next, node S6 at level number 4 is scheduled. Since node S6 does not require variables a, c and e these variables are dropped and are not carried when moving from node S4 to S6. Only variable b and variable d are carried. Node S7 is scheduled next. Since there are uses of variables b and d in S7, both are carried but variable g is in S6 is not carried. Lastly node S8 is scheduled. Since node S8 requires variable g it is carried to it from S7. S8 also needs variable f thus, aglet visits PE6.gatech.edu where f is located and fetches it. Thus, the effective migration schedule (from PE to PE) obtained for aglet under TOND is

PE1->PE2->PE5->PE3->PE1->PE3->PE4->PE5->PE4->PE6->PE7->PE8->PE8.
In case, of TALD, the effective migration schedule is the same as the schedule of nodes of the DDG since all the needed variables are carried and no fetches are required. Thus, in case of TALD the PE to PE schedule is:
PE1- >PE2- >PE5- >PE3- >PE4- >PE6- >PE7- >PE8.
As we see that in case of the TOND policy, the maximum amount of data movement in during a migration is 9,000 bytes. However in this case number of migrations increase to 13 while in case of the TALD we have only 7 migrations as seen from the above schedule.

4 Results

4.1 Outline of Implementation

We now describe our compiler framework for optimizing Java Mobile Agents. The input to our compiler contains only the data distribution and computation in form of class file which has to be performed. The output contains the aglet source code of the desired schedule along with generated code to migrate with variables from one scheduled computation to the next. Wherever needed, the code is generated for migrating the aglet to fetch remote data (under TOND policy as shown in code generation algorithm).
The various phases of the framework are given in figure 3.

4.1.1 Pre-Compiler Phase

In this phase annotated Java program is scanned and computation to be performed at each node is kept in table called computation table. As discussed earlier, we use ‘owner computes’ rule currently to decide which computation should be scheduled where.

4.1.2 Pre-Optimization Phase

In this phase, the annotated Java program is scanned to identify the annotations and also perform the normal frontend phases like scanning, parsing and semantic analysis. Using the annotations a table, identifying the variables that hold distributed data, henceforth called Distributed Data table (DDT) is also generated. At the end of the phase, an Abstract Syntax Tree of the program is obtained as output.

4.1.3 Optimization Phase

In this phase, the AST and the other tables are used to generate the Data Dependency Graph. Using these graphs the best schedule of the mobile agents is obtained by using the algorithm described in the previous section.

4.1.4 Agent Generation Phase

In this phase, the aglet source of the input Java program is generated. The code generation algorithms described in the previous section are used to generate the sets of variables to be carried during each migration. The computation to be performed at each processing node is wrapped inside a function call, and, using the Aglets class library, calls are generated for migration and for carrying variables.

4.1.5 ByteCode Generation Phase:

This is the final phase wherein the Aglet files generated are compiled using a Java compiler to generate the bytecode of the aglets. Since the mobile agents generated are based on set of APIs developed using Java, they can be compiled using any Java compiler following the Java Language Specification and can be run on any Java Virtual Machine. In order to run the mobile agents, all the machines should be running an agent server that listens, accepts and sends out mobile agents.

We use the Open Source Java compiler from IBM called Jikes. Figure 4 illustrates the phases modified and the new phases introduced in our approach to create the mobile agents. The new phases introduced are shown in dashed lines. The dashed arrows indicate the flow in our compiler while the solid arrows show the flow in Jikes.

Finally, the mobile agent package we used for generating the
mobile agent is the Aglet Software Development Kit from IBM, Japan.

4.2 Performance Evaluation

We tested our framework using several distributed applications and micro benchmark. In this section we give the results of our experiment with different strategies.

Table 2 gives the results for the effect of different scheduling policies on TALD for the micro benchmark used in the illustrative example of section 3.3. As one can see, the generated schedule performs better under TALD than other schedules. This is due to the fact that by scheduling nodes which have lower data requirement first, under TALD, the amount of data carried during each migration is kept at its lowest. Thus, one can see that during migrations #5, #6 and #7, our schedule carries less amount of data under TALD giving a better performance.

For the same example benchmark, TOND performs worse than TALD (refer to Table 3). This is due to higher number of migrations in TOND compared to TALD and as one can see that the reduction in maximum data carried during a migration does not have a significant effect on performance especially due to relatively small data sizes involved.

Table 4 illustrates the performance results for the database benchmark query [12]. In the database query as shown in figure 5 we find the salary of engineers who are currently working on CAD project, who are below a certain age and who were working on CAM project. Station A holds a database containing project name and it's corresponding number. Station B contains two databases: present assignment database and previous assignment database. Present assignment database contains project numbers and list of employees working on them. Previous assignment database contains project numbers and list of employees who were working on them. Station C holds a database containing employee id, age and title. Station D contains a database containing title and pay. Thus, to compute the above query mobile agent will have to go to PEs where databases are distributed and perform necessary database operations.

As we can see from table 4 that for 10,000 employees, the maximum amount of data transferred during a migration by TOND reduces by 36% at the cost of twice in number of migrations. Execution time increased by 25% for the TOND strategy. This is because of the increase in the number of migrations. However when we increased database size to 150,000 employees we achieve a speed up of 4% due to TOND strategy over TALD strategy (please refer to table 5. For a database of size 200,000 TOND outperforms TALD by a factor of 6% or so (refer to table 6). Since in TOND only a small amount of data has to be migrated in each migration the serialization time is very small. This drastically increases for TALD where large amount of data has to be transferred. The total number of migrations (and hence # serializations) is less in case of TALD. The migrations of large data are quite expensive and hence, in case of large size databases TOND does better, whereas for smaller size databases, TALD does better.

Finally, we decided to compare our optimizations using typical operations from data mining applications generating random schedules on them. In data mining basic operations are gather, fusion, consistency checks and merge sort. In table 7 we compare the timing difference between randomly generated schedules for each of these basic operations and the schedule generated by our method along with the TALD strategy. The data-bases of sizes ranging from 40000 to 4000 were used distributed over 10 PEs. The different operations tested were as follows:

Gather() In the gather operation we visit a node and pick all the data available at that node and move to the next node.

Fusion() In this operation, we fuse two data sets available at two different PEs. Thus, we visit a node, leave our data there and pick up all the data at that node and bring it back.

Consistency_Check() In this we visit a node, check if the data carried by the agent and data present locally on a PE are equal.

Merge_Sort() In this operation we visit a node, merge the data present at node with the already carrying data, sort this data and carry this to next node.

Due to our techniques, we achieved a speed-up of around 10% for each of these operations. This speed-up is due to
### Table 2: Comparison of different schedules for the micro benchmark under TALD

<table>
<thead>
<tr>
<th>Schedules</th>
<th>1 (KB)</th>
<th>2 (KB)</th>
<th>3 (KB)</th>
<th>4 (KB)</th>
<th>5 (KB)</th>
<th>6 (KB)</th>
<th>Total (KB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1S2S3S4S5S6S7S8</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>17</td>
<td>23</td>
<td>84</td>
</tr>
<tr>
<td>S1S2S3S4S5S7S6S8</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>19</td>
<td>21</td>
<td>84</td>
</tr>
<tr>
<td>S1S2S3S4S5S7S6S8</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>19</td>
<td>21</td>
<td>82</td>
</tr>
<tr>
<td>S1S2S3S4S5S6S8</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>19</td>
<td>76</td>
</tr>
</tbody>
</table>

### Table 3: Comparison of TALD and TOND for micro benchmark

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Max. Data During A Migration</th>
<th>Total Migrations</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TALD</td>
<td>19KB</td>
<td>7</td>
<td>11.6</td>
</tr>
<tr>
<td>TOND</td>
<td>9KB</td>
<td>13</td>
<td>15.2</td>
</tr>
</tbody>
</table>

### Table 4: Comparison of TALD and TOND for Database of 10000 employee

<table>
<thead>
<tr>
<th>Query</th>
<th>TALD</th>
<th>TOND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max. Data in a Migration (KB)</td>
<td>Migrations</td>
</tr>
<tr>
<td>Age&lt;30</td>
<td>45.56</td>
<td>3</td>
</tr>
<tr>
<td>Age&lt;35</td>
<td>46.20</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 5: Comparison of TALD and TOND for Database of 150000 employee

<table>
<thead>
<tr>
<th>Query</th>
<th>TALD</th>
<th>TOND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max. Data in a Migration (KB)</td>
<td>Migrations</td>
</tr>
<tr>
<td>Age&lt;30</td>
<td>587.3</td>
<td>3</td>
</tr>
<tr>
<td>Age&lt;35</td>
<td>593.2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 6: Comparison of TALD and TOND for Database of 200000 employee

<table>
<thead>
<tr>
<th>Query</th>
<th>TALD</th>
<th>TOND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max. Data in a Migration (KB)</td>
<td>Migrations</td>
</tr>
<tr>
<td>Age&lt;30</td>
<td>672.3</td>
<td>3</td>
</tr>
<tr>
<td>Age&lt;35</td>
<td>679.4</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 7: Timing difference between randomly generated schedules and compiler generated schedules

<table>
<thead>
<tr>
<th>S.No</th>
<th>Distributed Data Applications</th>
<th>Randomly generated Schedule (Sec)</th>
<th>Compiler generated Schedule with TALD (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gather</td>
<td>22.396</td>
<td>20.24</td>
</tr>
<tr>
<td>2</td>
<td>Fusion</td>
<td>26.32</td>
<td>22.34</td>
</tr>
<tr>
<td>3</td>
<td>Consistency Chk.</td>
<td>32.12</td>
<td>28.31</td>
</tr>
<tr>
<td>4</td>
<td>Merge Sort</td>
<td>49.21</td>
<td>44.18</td>
</tr>
</tbody>
</table>
our scheduling policy along with TALD optimizes the data movement reducing migration overheads over the randomly generated unoptimized schedules.

5 Related Work

Although there are different Java based mobile agent toolkits available, they rely entirely on the ability of the users to write mobile agent based distributed applications.

Traveler [18] is a Java-based mobile agent infrastructure, which supports wide area parallel applications. It allows the clients to declare the parallel applications as agents. These agents are dispatched through a resource broker. The resource broker executes it on the parallel virtual machine formed across the available servers. The mobile agents can move across the servers for load balancing, enhancing data locality and fault tolerance. Traveler uses RMI and object serialization. But this relies on the user’s ability to program using the Traveler toolkit and identifying the Agent Task.

StratOSphere [19] is a framework which distributes processing across several host systems and supports mobile Java applications. It addresses the issue of accessing distributed data and also distributing the computation. Here again the user is responsible and, as we have shown, compiler generated policies and schedules optimize the performance of agents better even over hand written random schedules. Parallel, distributed and mobile Java applications can be written using Ajents [8], which is a set of class libraries written in Java. Ajents are created using RMI and allow mobility of remote objects. They checkpoint (make a copy of the object state) using serialization techniques. When the Ajent is moved from one machine to another, it starts continuing from the latest checkpointed state. Though Ajents allow for easy creation of objects on remote hosts and interaction with them, they again are a set of libraries which the user can make use of to create mobile Java applications.

Jaspal Subhlok et al [16], present a solution to the problem of node selection for executing parallel and distributed applications. They provide a solution for automatic selection of nodes for executing a performance critical parallel application in a shared network. They also introduce an algorithm for analyzing computation and communication resources together for different applications. But they do not deal with the issue of serialization overhead and its minimization which is critical to the performance of Java agents.

Javelin [4] presents a Java based technology which uses the Internet for global computing. It aims at using the unused computer resources across the Internet for doing useful computation. Javelin consists of Clients that seek computing resources, Hosts that offer computing resources and Brokers that coordinate the supply and demand of computing resources. Clients that require computing resources submit request to the broker. The Broker based on the availability of Hosts allocate computing resources to the task submitted by the client. This requires only a web browser to be present and no other specialized software. Though this achieves wide area parallel computing in a more oriented towards balancing the large parallel process across the available computers. Javelin has been extended to Javelin++ [13]. Javelin++ uses Java applications instead of Java applets (as used by Javelin) and is much more scalable than its predecessor. The paper also discusses some schedulers and compares the performance of the schedulers. Javac [3] is a restructuring Java Compiler which takes in an annotated code which identifies those code that can be scheduled in parallel in a multiprocessor environment. It makes use of Java threads to achieve parallelism. It reads the annotations and identifies the number of threads to be created and other such details and creates that many threads. It targets a multiprocessor environment and not a distributed one. Straker et al [15], describe an agent framework for transparent distribution and concurrent execution of computations. It splits the provided computation to number of work packages and then a coordinator distributes all these to several worker agents. The worker is associated with a strategy, which is obtained as user input. The worker will migrate to some place where it computes the results and returns. This paper identifies the job distribution based on the availability of workers and creates new workers if no other worker needs the strategy in question and if there are no other workers free. Aqbal et al [2] illustrate that good performance can be extracted from the agent based strategy by critical sequence of mixed remote procedure calls and agent migration. They have evaluated the cost of migration of the agents to different destinations, when the communication sequence is fixed and when it is not fixed(tree structured and series-parallel graph). They find the shortest path between the start node and the end node and equate it to the optimal migration sequence. The size of data being operated on is not considered and this work is more oriented towards reducing the distance traveled by an agent rather than minimizing the data carried. J Bredin et al [5] explain about a scheduling scheme for mobile agents in a particular application environment and does not discuss scheduling of mobile agents in general to minimize the data carried around.

In contrast to all the works mentioned above, this paper provides a way of analyzing an annotated Java program to schedule the Mobile agents efficiently. It proposes two scheduling strategies Take All Live Data (TALD) and Take Only Needed Data (TOND). The mobile agent is scheduled by either of these strategies so that it either carries as thin data as possible during a migration or it makes minimum number of migrations, ensuring that the schedule formed is legal and preserving the dependencies in the program.

6 Conclusion and Future Work

As shown in the paper, compiler based analysis helps in reducing the agent migration overheads and improves performance. We have provided two strategies TOND and TALD, which reduce the overheads for large data sizes and small data sizes respectively trading the number of migrations vs. the amount of data carried during a migration. We have shown that, in general, compiler generated schedules coupled with the use of these policies result in an efficient agent code. This work will have the potential of impact on different research communities. For database researchers we provide a powerful tool for performing distributed database computation. We provide a very high level of abstraction for mobile code generation. Given a data base distribution, size of databases and query our compiler scheduler generates efficient agent code. For compiler researchers we provide a novel use of advanced compiler techniques for efficient migration of mobile code. For network and mobile agent researchers this work proposes a new direction of research.

In this work, we have assumed a simplistic rule to distribute the computation using ‘owner-computes’. We assume that the size of databases and dependencies are known at compiler
time. We also do not look into security aspects of mobile code for distributed data mining applications. This work is an attempt towards performing computation distribution to induce effective minimization of both the number of migrations and total overhead of each migration in mobile code for distributed data mining applications by using advanced compiler techniques. As future work we plan to move our system to take these decisions at run time. Research in mobile agent is growing. As more applications of mobile agent technology emerge, we plan to extend our work to them.

Acknowledgments

We are grateful to Sham Navathe for providing us with useful insight on Distributed Database Systems.

References


APPENDIX

double[] a://b(a,machine1.gatech.edu,40000);
double[] b://b(b,machine2.gatech.edu,36000);
double[] c://b(c,machine3.gatech.edu,32000);
double[] d://b(d,machine4.gatech.edu,28000);
double[] e://b(e,machine5.gatech.edu,24000);
double[] f://b(f,machine6.gatech.edu,20000);
double[] g://b(g,machine7.gatech.edu,16000);
double[] h://b(h,machine8.gatech.edu,12000);
double[] i://b(i,machine9.gatech.edu,8000);
double[] j://b(j,machine10.gatech.edu,40000);

Randomly Generated Schedules

Procedure Gather(){
    for(int i=0;i<10;i++)
    {
        x = Random_schedule[i];
        Goto(x);
        Carry(var);
    }
}

Procedure Fusion(){
    for(int i=0;i<10;i++)
    {
        if(i==0)
        {
            x = random_schedule[i];
            Goto(x);
            result = var_at_x;
            Carry(result);
        }
        else
        {
            x = random_schedule[i];
            result = Fuse(result,var_at_x);
            Carry(result);
        }
    }
}

Procedure Consistency_chk(){
    for(int i=0;i<10;i++)
    {
        if(i==0)
        {
            x = random_schedule[i];
            Goto(x);
            result = var_at_x;
            Carry(result);
        }
        else
        {
            x = random_schedule[i];
            Check(result == var_at_x);
            result=Fuse(result,var_at_x);
        }
    }
}

Procedure Merge_Sort(){
    for(int i=0;i<10;i++)
    {
        if(i==0)
        {
            x = cs_schedule[i];
            Goto(x);
            result = var_at_x;
            Carry(result);
        }
        else
        {
            x = cs_schedule[i];
            result = Merge(result, var_at_x);
            Sort(result);
            Carry(result);
        }
    }
}

CS generated Schedules

Procedure Gather(){
    for(int i=0;i<10;i++)
    {
        x = cs_schedule[i];
        Goto(x);
        Carry(var);
    }
}

Procedure Fusion(){
    for(int i=0;i<10;i++)
    {
        if(i==0)
        {
            x = cs_schedule[i];
            Goto(x);
            result = var_at_x;
            Carry(result);
        }
        else
        {
            x = cs_schedule[i];
            result = Fuse(result,var_at_x);
            Carry(result);
        }
    }
}

Procedure Consistency_chk(){
    for(int i=0;i<10;i++)
    {
        if(i==0)
        {
            x = cs_schedule[i];
            Goto(x);
            result = var_at_x;
            Carry(result);
        }
        else
        {
            x = cs_schedule[i];
            Check(result == var_at_x);
            result=fuse(result,var_at_x);
        }
    }
}

Procedure Merge_Sort(){
    for(int i=0;i<10;i++)
    {
        if(i==0)
        {
            x = cs_schedule[i];
            Goto(x);
            result = var_at_x;
            Carry(result);
        }
        else
        {
            x = cs_schedule[i];
            result = Merge(result, var_at_x);
            Sort(result);
            Carry(result);
        }
    }
}