A Multistep Approach for Restructuring and Mapping Distributed Object-Oriented Software onto a Multiprocessor System

Safwat H. Hamad  
Faculty of Computer & Information Sciences,  
Ain Shams University,  
Abbassia, Cairo,11566,  
Egypt  
Safwat@engr.uconn.edu

Reda A. Ammar  
Computer Science & Engineering Dept.,  
University of Connecticut, Storrs, CT,  
06269, USA  
Reda@engr.uconn.edu

Mohammed E. Khalifa  
Faculty of Computer & Information Sciences,  
Ain Shams University,  
Abbassia, Cairo,11566,  
Egypt  
esskhalifa@yahoo.com

Ayman El-Dessouky  
National Authority of Remote Sensing and Space Sciences,  
Cairo, Egypt  
ayman@eri.sci.eg

Abstract

A large number of Distributed Object Oriented (DOO) systems have been developed for solving complex problems in various scientific fields. In a distributed object-oriented application, classes can run on a separate computer within a network system. So, they should be distributed efficiently among different nodes. However, the initial design of the DOO application does not necessarily have the best class distribution. In such a case, the DOO software may need to be restructured. The challenge resides in the complexity of interactions between objects.

In this paper, we present an approach for efficiently restructuring the DOO software classes in order to be mapped onto target distributed architecture. The proposed methodology consists of a set of consecutive steps. In the first step we are using the Distributed Object-Oriented performance (DOOP) model as a technique for the assessment of relationships between system classes. Next, a recursive graph clustering technique is utilized to partition the OO system into subsystems that have low coupling and are more suitable for distribution. Then, the mapping process is applied to map the generated partitions to the nodes of the target architecture. An experimental case study is then studied to illustrate the results after applying each step of the proposed approach.

1. Introduction

Object-oriented techniques gained popularity either in research or in commercial applications. A program is organized as a set of interacting objects, each of which has its own private state rather than a set of functions that share a global state. Classes represent abstraction that makes adapting software easier and thus lower the cost of reuse, maintenance and enhancement. Object-oriented paradigm is based on several concepts such as encapsulation, inheritance, polymorphism, and dynamic binding [4, 5]. Although these restructuring DOO applications and then mapping the resultant modules to a specific multiprocessor architecture. The presented process is achieved in a set of consecutive features contribute to the reusability and extensibility of systems, they produce complex dependencies between classes. The most interesting feature of OO approach is that objects may be distributed and executed either sequentially or in parallel [2].

A distributed OO application consists mainly of a set of interacting objects; each one runs on a separate computer within a network system. There have been a large number of projects related to the use of object-oriented approaches for the design of problem solving environments for complex applications in various scientific fields [7]. The advent and wide adoption of the Java(tm) programming language and environment has provided an opportunity to develop object-oriented techniques on network environment [1].

Designing a DOO application involves an important phase to tune performance by “concretizing” object locations and communication methods [4]. At this stage, it may be necessary to use tools to allow analysis of the communication patterns among objects in order to take the right allocation decision. Another important issue to consider is the mismatch problem between the software structure and the available hardware organization [6, 7, and 8]. In such class of problems, the solution is possible through two approaches: either to configure the hardware to match the software components (hardware reconfiguration), or to reconfigure the software structure to match the available hardware by reorganizing its components (software restructuring).

It has been shown that performing software restructuring ahead of the allocation and scheduling phases improved the results obtained from these phases and reduced the overall resources cost [7, 8]. Unfortunately this technique is not applicable for DOO systems. In DOO systems, the intensive class interaction results in a communication cost due to remote function call. Furthermore, the dependency between classes (including composition and inheritance) results in what is called “dependency communication”.

In this paper, we introduce an efficient approach for steps as illustrated in figure 1. In the first step, the Distributed Object Oriented Software (DOO) is analyzed using DOOP model in order to evaluate of relationships and
interactions among system classes. The resultant inter-class communication costs are modeled as a class dependency graph. The second step is concerned with identifying clusters of a dense community of classes within the DOO system. This is done through a recursive graph clustering technique that decomposes the system into subsystems that have low coupling and are suitable for distribution. Next, the generated clusters are merged into groups. Those groups are then mapped onto the nodes of the multiprocessor system such that the amount of communication among classes is minimized.

Figure 1. The process for restructuring and mapping DOO software.

The rest of this paper is organized as follows: section 2 describes how to model interclass communication using undirected graphs while incorporating the DOOP model to measure the communication cost between different classes. Then each phase of the presented multistep approach is illustrated in a separate section: section 3 outlines the clustering phase, section 4 addresses the step of cluster grouping, and section 5 discusses the mapping problem. A case study is discussed in section 6 illustrating the implementation of the presented approach in a simulated environment. Finally, comes the conclusions.

2. Representing Class Dependency

Performance analysis is an important phase of the software development process. It does not only evaluate the resources utilization but also will provide a way to compare different design alternatives, detect bottlenecks, maintain and re-use software as well as identifying the tradeoffs. Classical techniques and methodologies of performance analysis are either unsuitable or unnatural to capture performance behavior of Object-Oriented (OO) systems. However, The Performance-Based Model for Distributed Object-Oriented Software (DOOP) can.

The DOOP [5] introduced an analytical modeling approach that analyzes and evaluates Distributed Object Oriented Systems. According to the model, each node in a DOO system is represented in hierarchical levels of abstraction. Hence it can be used as a powerful analytical tool to evaluate the overall cost of the communication activities that exists among classes as described in [2]. These costs are then used to generate the Class Dependency Graph (CDG).

The class dependency graph CDG as shown in Figure 2 is a graph representation for interclass communications. In CDG, each class is represented as a vertex and the communication between two classes is represented by an undirected weighted edge. For example, an edge between class A and B represents a communication activity that exists between these two classes. The weight of the edge WAB represents the cost of that communication activity. If no data communication or relationship dependency has been recognized in the OO application between two classes, no edges will connect them in the CDG.

Figure 2. Graph representation for interclass communication.

3 Clustering System Classes

In this section, we describe a clustering technique that is considered the first primary phase of the restructuring approach presented here. After applying this step, the object oriented system is decomposed into subsystems that have low coupling and are suitable for distribution. It is assumed that the OO system is already represented as a Class Dependency Graph (CDG).

The technique is based on a recursive use of a spectral graph bi-partitioning algorithm to identify dense communities of classes. At each recursive call, the CDG is partitioned into two sub graphs each of which will be further bi-partitioned as long as the partitioning results in clusters that are denser than their parents. The iteration stops when at least one of the produced sub-graphs is badly partitioned. A sub-graph is considered Well-Partitioned if the summation of the weight of internal edges (those between the vertices within a sub-graph) exceeds those of external edges (those between the vertices of the sub-graph and all the other vertices in other sub-graphs). In this case, the bi-partitioning
step is considered obsolete and the algorithm will backtrack to the clusters generated in the previous step.

```
ALGORITHM GraphCluster(C, Clusters, ClIndx)
INPUT: C = Adjacency matrix (weights matrix).
       Clusters = a vector indicating the cluster no. of each node in the CDG graph.
       ClIndx = the cluster index representing the subgraph needed to be bi-partitioned.
OUTPUT: NewClus = a vector indicating the new cluster numbers to which each node in the graph belongs.

STEP 1 Let CurrC be the extracted Adjacency matrix of the graph indicated by ClIndx.
STEP 2 Partition the CurrC into two subgraphs
       Indx = GraphBipart(CurrC)
STEP 3 Create vector G1 and G2 to hold the indices of the first and the second subgraphs respectively.
STEP 4 IF ( NOT WellPartitioned(CurrC,G1)) OR ( NOT WellPartitioned(CurrC,G2) ) THEN
       return Clusters
END IF
STEP 5 Update NewClus with new portioning in G1 & G2.
STEP 6 Recursively bipartition G1 and G2
       NewClus = GraphCluster(C, NewClus, ClIndx*10+1)
       NewClus = GraphCluster(C, NewClus, ClIndx*10+2)
```

**Figure 3. A recursive graph clustering algorithm.**

```
ALGORITHM K-Partition(C, k)
INPUT: C = nxn Adjacency matrix (edge-weights matrix).
       k = the number of required partitions
OUTPUT: Indx = an n-vector with the cluster index for every node

STEP 1 IF C is not doubly stochastic THEN
       Sum every row in C
       Let Max be the largest row sum
       Normalize C = C/Max
END IF
STEP 2 Compute largest k eigenvalues and eigenvectors U and R respectively
STEP 3 Let U’ be the transpose of U
STEP 4 Compute the matrices Q and R from the QR-factorization with pivoting of U’
STEP 5 Let R1 be the first k columns in R
STEP 6 Compute Z = Q(R1’)^-1
STEP 7 Compute Π = UZ
STEP 8 Set Indx to be the indices of the max value in each row of Π
STEP 9 Return Indx
```

**Figure 4. A Spectral k-partitioning Algorithm.**

At the end, the identified sub-graphs are the suggested clusters of the system. The strategy of breaking the iteration has been found to be very useful as the algorithm does not stop when a certain predefined number of clusters are composed. Figure 4 shows a detailed step by step description of the clustering Algorithm. The mathematical derivation of the spectral factorization algorithm used in our clustering approach appears in [6]. It is originally solving the I-bounded Graph Partitioning (I-GP) problem. However, we have adapted the solution methodology to fit within our bi-partitioning approach. More details can be found in [2].

The result of this step will be a cluster graph, in which the nodes represent clusters. Inside every cluster there exist several classes which have zero intra-communication overhead. As far as clusters intercommunication is concerned, the edges will capture summation over all the classes’ intercommunication overhead for all clusters.

### 4 Grouping Clusters

In this step, the restructuring process is accomplished by merging the generated system clusters into groups in order to prepare them for the upcoming mapping step. The mapping process has two cases. The first case appears when the number of candidate clusters are less than or equal to the number of the available nodes. In this case the mapping process will be done simply by assigning each cluster to one of the available nodes. The problem occurs in the second case, when the number of the generated clusters exceeds the number of available nodes. This is a more realistic view since there will always be huge software systems and limited hardware resources.

The grouping process can be done in various ways. The authors have already investigated a number of approaches and discussed them in [3]. The objective is to merge clusters into groups in such a way to keep the communication costs among them minimized. In this paper we employ a technique that uses the cluster graph generated at the previous step and group it into a Merged Cluster Graph (MCG).

To achieve this we propose using the K-Partitioning algorithm [8] to partition the cluster graph. The K-Partitioning algorithm partitions the input graph into a pre-specified number of sub-graphs. This number is determined by the value of the parameter k (as shown is figure 4). Here, we apply the K-Partitioning algorithm on the cluster graph to generate a set of groups whose number equals exactly to the number of nodes in the target system. Now, the resultant MCG can be directly mapped to the multiprocessors system so that the inter-cluster communication cost is minimized.

### 5 Physical Mapping

The process of efficient placement of the MCG into a real processor network topologies is called physical mapping. Most of the research work on mappings considered structured topologies, including hypercube, mesh or ring architectures. However, an increasing number of applications demand methods dealing with arbitrary processor interconnection topology. The general physical
mapping problem of an arbitrary graph with communication delays onto a fixed size and connection pattern distributed architecture was proven to be NP-hard, thus allowing only for heuristic approaches [8].

For the work presented here, we have employed the mapping algorithm presented in [7]. It is called the (PMA), where the target distributed architecture was modeled by a non-directed graph with unitary edge weights only. The assignment procedure starts by placing the highly communicative clusters on adjacent nodes of the processor network. Once these backbone clusters are mapped, there is no backtracking, thus achieving low complexity. For each of the backbone clusters, the algorithm creates sets of neighboring clusters which are candidate for placement into the adjacent nodes of every pre-assigned backbone cluster. The algorithm ends when all clusters are assigned.

Different mappings can be evaluated using the following cost function:

\[
\text{CF}(F_m) = \sum_{(v_u, u_v) \in E_p} \text{dist}(F_m(v_u), F_m(u_v)) \times \text{comm}(v_u, u_v)
\]

Where:

- \(G_c(V_c, E_c)\) is graph of clustered classes,
- \(G_p(V_p, E_p)\) defines the graph of physical processors,
- \(F_m: V_c \rightarrow V_p\) is the physical mapping function,
- \(\text{dist}(F_m(v_u), F_m(u_v))\) is the shortest path in the processor graph between the 2 processors-nodes where the tasks \(v_u\) and \(u_v\) are mapped, and
- \(\text{comm}(v_u, u_v)\) is the total communication cost between the tasks \(v_u, u_v\)

A mapping \(F_m\) with respect to Cost Function \(\text{CF}()\) is called optimal if:

\[
\text{CF}(F_{m_{opt}}) = \min \{ \text{CF}(F_m) \mid F_m \in \text{MAP} \}, \text{ where MAP is the set of all possible mappings}
\]

6. A Case Study

In this section we are going to provide a step-by-step illustration of applying the proposed restructuring and mapping technique presented above. This is done by means of a case study. We have developed a performance-driven restructuring simulator using Matlab 7.0.4. A Graphical User Interface (GUI) utility has been implemented to show the system status after each step while the algorithm proceeds. The simulator has a friendly user interface that allows the user to specify the nodes and edges in the systems, and then it will be able to generate the class dependency graph (CDG).

We conducted an experiment using an OO system that consists of 28 classes. The CDG that was generated by the simulator for this system is given in figure 5. Figure 6 shows the resultant system clusters generated by the proposed bi-partitioning algorithm. We can see that the proposed algorithm has created 7 clusters each of which is marked up with a different color and surrounded by an oval for further highlight.

Furthermore, let us assume that the target distributed environment consists of 4 homogenous nodes that are connected in a ring topology. As shown in figure 7, applying the Cluster Grouping step generated 4 large groups. Now we can model this grouping as a MCG as shown in figure 8. Applying the PMA on the resultant MCG resulted in assigning g1 to P3, g2 to P1, g3 to P2, and g4 to P4. Obviously, we can directly substitute those groups by the classes they represent. This means that the P1 will have \{C9, C10, C11, C12, C21, C22, C23, C24\} allocated to it. In addition, \{C13, C14, C15, C16\} allocated to P2, \{C1, C2, C3, C4, C5, C6, C7, C8\} to P3, and \{C17, C18, C19, C20, C25, C26, C28\} to P4. The Mapping cost was computed as follows:

\[
\text{CF(PMA)} = 91 + 74 + 57 + 2 \times 43 = 308.
\]
7 Conclusions

In this paper, we presented an approach for efficiently restructuring and mapping DOO applications to a multiprocessor architecture organized in any generic topology. The proposed methodology consists of a set of consecutive steps. First, the DOOP model was used efficiently to evaluate the communication costs between different classes. Next, the OO system was partitioned into subsystems that have low coupling. Then, the identified subsystems were merged into larger groups. Those groups were assigned to the nodes of the target architecture using a low complexity mapping algorithm. The whole approach steps were implemented and tested over a set of case studies. Future work includes the introduction of different mapping strategies which would improve the performance of the mapping step over some other existing heuristics.

8 References