Partitioning and Allocation of Objects in Heterogeneous Distributed Environments Using the Niched Pareto Genetic-Algorithm

Seunghoon Choi and **Chisu Wu**
Department of Computer Science
Seoul National University
Kwanak-Gu, Seoul, Korea
{csh, wuchisu}@selab.snu.ac.kr

Abstract

As the importance of middleware-based distributed object computing environments (e.g. CORBA and DCOM) increases, there is considerable interest in incorporation of object-orientation (OO) and distributed systems. One important aspect of distributed object systems is effective distribution of software components, to achieve some performance goals, such as balancing the workloads, maximizing the degree of concurrency and minimizing the entire communication costs. Although there have been a lot of works on partitioning and allocation for distributed system, they are not directly applicable to OO system. We developed a partitioning and allocation model for mapping OO applications to heterogeneous distributed environments, and evaluated it using genetic algorithm (GA). Our model applies the graph-theoretic approach, dealing with a lot of characteristics of OO paradigm. The Niched Pareto GA [7] is adopted to experiment our model because a partitioning and allocation problem is multiobjective problem with non-commensurable objectives.

1. Introduction

Recently, there is considerable interest in distributed software development. One important aspect of distributed systems is effective distribution of software components, to achieve some performance goals, such as minimizing the execution time, balancing the workloads, and minimizing the communication costs. Partitioning is the process of splitting the application into components, and allocation is the process of allocating those components to physical nodes. Partitioning and allocation is an essential part to effective distribution of software components.

As the importance of middleware-based distributed object computing environments (e.g. CORBA and DCOM) increases, incorporation of object-orientation and distributed systems became a principal research topic in academic and industrial area. From the viewpoint of software engineering, object-oriented (OO) paradigm provides a concept as effective as that provided by the distributed process communication model: objects communicate via method invocations, and distributed processes communicate via remote procedure calls(RPCs) [4].

There have been a number of works on partitioning and allocation for the distributed and parallel programs. Although they provide several useful techniques and metrics for partitioning and allocation, these works are not satisfied in terms of object-based distributed environments. The reason is that the distribution unit of their model is a process, task, job or module that has numerous different properties from object. A process or task has completion time, and there is partial ordering among processes to complete their functions. An object has no completion time and there is no partial ordering among objects, although there is execution sequence to perform some services among each method of the objects. In addition conventional process clustering for a distributed programming environment just minimize the overhead of routing messages [2]. An appropriate approach to group distributed objects must address the inherent characteristics of object system model, such as the communication cost between objects, the cost of invoked method execution, class hierarchy, dynamic binding, object oriented language constructs, the method invocation frequency, workload of each object, and so on.

Various approaches to the object grouping and allocation have been proposed. Clustering techniques in DIAMONDS system [3] groups the fine-grained active objects for the purpose of minimizing communication and run-time overhead while still realizing the full potential for concurrency intrinsic to active object designs. Unit of distribution in this method is a group of objects called cluster. Cost function for clustering is a simple summation of communications cost and concurrency. We split these into two factors and consider them independently. In addition, our approach considers the workload of each object and the system architectures.

A clustering strategy in message-passing systems proposed in [2] [4] is a complete-linkage hierarchical clustering approach considering both communication overhead and the degree of similarity on method codes. Its cost function of one method invocation consists of the number of messages to reach the destination and back to source and the number of messages for further invocations of method execution. This technique tried to minimize the cost of invoked method execution on the associated class hierarchy, but doesn't address concurrency.

In [5], metrics and techniques for automatic partitioning and assignment of object-based concurrent programs are described. This partitioning algorithm attempts to achieve a small amount of communications and a large degree of potential for concurrent execution. The granularity of partitioning is at the level of the Ada program unit (an object, a class, a task, a package or subprogram) and unit of distribution is the partition. We seek to develop the partitioning/allocation model where the unit of distribution is the object. In concurrency metric defined in this method, they didn't address the precise process to calculate the degree of potential parallelism while we tried to estimate the potential concurrency among the objects in legacy OO system.

In this paper we develop a partitioning and allocation model for mapping OO application to heterogeneous distributed systems, and evaluate it using Genetic Algorithm (GA) [13]. Our model applies the graph-theoretic approach dealing with numerous characteristics of OO paradigm. Additionally, individual metrics for communication cost and concurrency are proposed. Because a partitioning and allocation problem is multiobjective problem with non-commensurable objectives, our model is experimented by the Niched Pareto GA [7] [14].

This paper is organized as follows. Section 2 describes the problem definition, including our goal and several simplifying assumptions. In section 3, three main models, Method Dependence Graph (MDG), Class Dependence Graph (CDG), and System Graph (SG) are defined. Section 4 shows the results of the experiment of our approach using genetic algorithm and gives the associated analysis. Section 5 draws some conclusion and presents future works.

2. Problem Definition

We consider partitioning and allocating the C++ objects into heterogeneous distributed systems where a collection of autonomous computers with distinct computing power are interconnected through a communications network having different bandwidth. Partitioning and allocation of distributed objects is simply the choice of a mapping of a set of objects to a set of processors. To increase the overall performance of distributed OO systems there are a lot of goals, of which we pursue the following three goals:

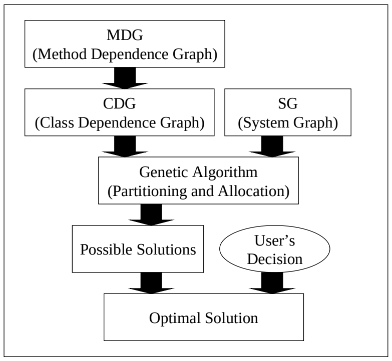


Figure 1. Overall Process of Partitioning and Allocation

- minimize the amount of communication between objects
- maximize the degree of parallelism for executing distributed object-oriented programs
- achieve the balanced distribution of workloads

To achieve above goals, the cost function must be defined treating the constraints and characteristics of distributed object-orientation paradigm, including the class hierarchy, dynamic binding, object oriented programming language (OOPL) constructs, the frequency of each method, the cost of method execution, the method invocation type (synchronous and asynchronous), and so on. The partitioning and allocation problem is addressed using a graph-theoretic representation of the program and the graph is partitioned so as to minimize the cost function.

Our model consists of three graphs, MDG, CDG and SG. MDG represents several characteristics of each method itself and interactions among methods. CDG represents several features of each class and interactions among classes and is constructed from MDG. SG represents the attributes of each machine sites and interconnection among computer or processors. CDG and SG are put as input into our genetic-based partitioning and allocation algorithm. The overall approach is shown in Figure 1.

To simplify the problem, a number of assumptions are made.

- All instances of the specific class are allocated and created in only one physical node, so the code migration to other nodes is not permitted during execution.
- All the sites have enough storage to accommodate any number of objects.
- The total number of classes is bigger than that of nodes or processors.
- For all methods in application, the size of the method in

the number of instructions executed can be estimated.

- All instances of the same class in the same node share the procedure code in order to save space, and the code of inherited method from a specific superclass is replicated into its descendant classes.

3. Models for Partitioning and Allocation

In this section, we propose the graph-theoretic models for partitioning and allocation of OO applications implemented in C++ and the metrics for communication cost and concurrency between objects.

3.1 Method Dependence Graph (MDG)

Although at the requirement analysis, design, and implementation stages the basic unit of OO paredigm is the object, in run-time the element of execution and communication is the method. Actually the function calls and communications between objects are performed through the method calls. So, before constructing the class relationship graph, the graph representing the relationships between the methods should be constructed.

MDG is an acyclic, directed and weighted graph, representing the relationships among the methods of the objects in application. Each method in the distributed OO application is modeled as a node and the method invocation between two objects is modeled by a directed edge. An edge is directed from the caller's method toward the callee's method. The weight of node denotes the size of the node in the number of instructions executed and the method invocation type (synchronous or asynchronous). The weight of arc indicates the network communication cost and the number of invocations of the method. MDG is denoted by:

MDG =(M, MR, MW, MIT, MIW, DMIF, TMIF, MPC)

- **M** (Methods) = {$c_i.m_j$}, where $c_i.m_j$ indicates method j of class i. If i = 0 then method j denotes a global function. If $i = 0 \wedge j = 0$ then $c_i.m_j$ denotes a main function.
- **MR** (Method Relationships) = {mr_{ikjl}}, where $mr_{ikjl} = 1$ if the direct invocation of method $c_j.m_l$ is made in the code of $c_i.m_k$. If pointer to c_j is defined and its virtual method m_l is invoked through that pointer by $c_i.m_k$, the value of mr_{ikjl} is 1. In addition, for all c_d, descendants of c_j, the value of mr_{ikdl} is 1. The reason is that the dynamic binding prevents us from determining which method is actually called in run-time. Otherwise $mr_{ikjl} = 0$.
- **MW** (Method Workloads) = {mw_{ik}}, where mw_{ik} indicates the size of $c_i.m_k$ in the number of instructions executed in run time.
- **MIT** (Method Invocation Type) = {mit_{ik}}, where $mit_{ik} = 1$ if $c_i.m_k$ is an asynchronous method, and $mit_{ik} = 0$ if $c_i.m_k$ is a synchronous method.
- **MIW** (Method Invocation Workloads) = {miw_{ikjl}}, where miw_{ikjl} indicates the network communication cost when $c_i.m_k$ calls $c_j.m_l$. If i = j, $miw_{ikjl} = 0$ because the network communication cost between two methods of same object is 0. In general, the network transmission time for a given data item depends on factors such as the size of the data item being transferred, the topology and bandwidth of the network, the type of switching used, the types of machine-network interfaces used and the overall network load. In our model, the network communication cost between two methods is represented by the number of packets of transmitted data.
- **DMIF** (Direct Method Invocation Frequency) = {$dmif_{ikjl}$}, where $dmif_{ikjl}$ indicates the number of direct invocations of $c_j.m_l$ that is made in the code of $c_i.m_k$. $dmif_{ikjl}$ is calculated by $mr_{ikjl}*(SN_{ikjl}+DN_{ikjl}+PN_{ikjl})$.

(1) SN_{ikjl} denotes the number of executions of $c_j.m_l$ through the static invocation made by $c_i.m_k$. This is calculated by

$$SN_{ikjl} = \sum_{s \in S_{ikjl}} P(if_s) \times N(loop_s)$$

, where S_{ikjl} is a set of statement in code of $c_i.m_k$ that calls statically $c_j.m_l$, $P(if_s)$ is the probability that the execution control reaches the statement s, and $N(loop_s)$ is the number of iterations of the statement s due to while-loop.

(2) DN_{ikjl} denotes the number of executions of $c_j.m_l$ through the dynamic invocation made by $c_i.m_k$. This is calculated by

$$DN_{ikjl} = \sum_{d \in D_{ikjl}} P(if_d) \times N(loop_d) \times P(dyn_d, c_j)$$

, where D_{ikjl} is a set of statement in code of $c_i.m_k$ that calls dynamically the virtual method $c_j.m_l$. $P(dyn_d, c_j)$ treats the dynamic binding and we assume that the virtual method of parent and its descendents has the same probability of execution in runtime. Generally $P(dyn_d, c_x)$ denotes the probability that $c_x.m_l$ is executed by the statement d ($d \in D_{ikjl}$) in runtime, calculated by 1/(1 + the number of descendents of c_j - $ACN(c_j)$). $ACN(c_j)$ denotes the number of abstract classes among c_j itself and all the descendents of c_j. The reason for subtracting $ACN(c_j)$ from denominator is that there is no instance of abstract class in runtime.

(3) PN_{ikjl} denotes the number of executions of $c_j.m_l$ due to the dynamic invocation of the virtual method m_l of one of c_j's ancestor classes made by $c_i.m_k$. This is calculated by

$$PN_{ikjl} = \sum_{c_p \in P_{ikjl}} \sum_{d \in D_{ikpl}} P(if_d) \times N(loop_d) \times P(dyn_d, c_j)$$

,where P_{ikjl} denotes a set of c_j's ancestor classes that include the virtual method m_j in its definition. If c_j is an abstract class, $PN_{ikjl} = 1$.

- **TMIF** (Total Method Invocation Frequency) = {$tmif_{ikjl}$}, where $tmif_{ikjl}$ indicates the total number of invocations of the method $c_j.m_l$ that is made by $c_i.m_k$ in execution time. $tmif_{ikjl}$ is calculated by

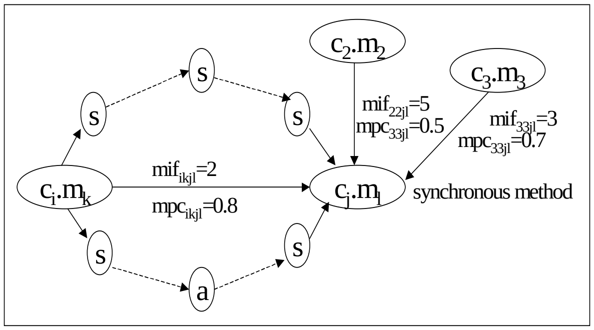
$$tmif_{ikjl} = dmif_{ikjl} \times \sum_{\forall c_x.m_y:mr_{xyik}=1} tmif_{xyik}$$

If x = 0 and y = 0, $tmif_{xyjl}$= $dmif_{xyjl}$ because $c_0.m_0$ is a main function. If j = 0 and l = 0, $tmif_{xyjl}$=1 because main function is called once. $\forall$ i, k, j, l : mr_{ikjl}=0, $tmif_{ikjl}$ = 0.

- **MPC** (Method Potential Concurrency) = {mpc_{ikjl}}, where the value of mpc_{ikjl} is between 0 and 1, where 0 denotes no concurrency and 1 denotes full concurrency. The concurrency metrics is obtained by extracting semantic knowledge that is implicit in the MDG. Processes are concurrent if they exist at the same time [6]. Because generally every pair of methods with no interaction is processed independently, we start to calculate the values of MPC after all elements of MPC are initialized to l. The concurency between the methods is dependent on whether the called method is invoked synchronously or asynchronously.

(1) Asynchronous: In asynchronous invocation, caller isn't bloked and continues to execute subsequent code. In this case, to allocate caller and callee in different site is better in terms of performance and completion time. So, the value of mpc is 1.

Figure 2. Concurrency in Synchronous Method Invocation



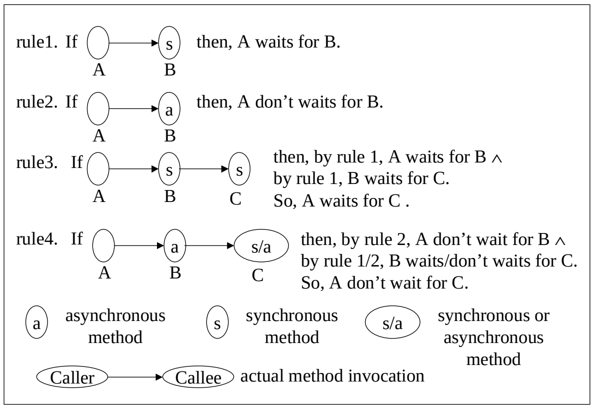


Figure 3. Waiting Rules for Synchronous and Asynchronous Method Invocation

(2) Synchronous: If one of two methods invokes a synchronous method, generally those two method cannot be executed concurrently because the caller shoud be blocked until the callee completes its execution. So the value of MPC is 0. But, there is an exceptional case where $c_i.m_k$ and $c_j.m_l$ are concurrent although in MDG there exist a synchronous call relationship between $c_i.m_k$ and $c_j.m_l$. For example, think about Figure 2 illustrating a part of MDG, where each of $c_i.m_k$, $c_2.m_2$, and $c_3.m_3$ has a synchronous call relationship with $c_j.m_l$. Although it seems that $c_i.m_k$ always waits for $c_j.m_l$ due to a synchronous call relationship, there is the case where $c_i.m_k$ and $c_j.m_1$ are executed concurrently because there is a case where $c_j.m_l$ is invoked not by $c_i.m_k$ but by $c_2.m_2$ or $c_3.m_3$ in run time. In addition to direct call from $c_i.m_k$ to $c_j.m_l$, all of the indirect calls from $c_i.m_k$ to $c_j.m_l$ should be considered to measure more accurate concurrency. To calculate the total number of indirect invocations of $c_j.m_l$ that $c_i.m_k$ makes with $c_i.m_k$ being blocked, two types of path in indirect call from $c_i.m_k$ to $c_j.m_l$ should be differentiated. First type is the path that has some asynchronous methods in the middle of it. Second type is the path that consists of only synchronous type methods. We should include only second type of path into mpc_{ikjl} due to the waiting rules in Figure 3. These rules indicate that in case of first path type having some asynchronous method in the middle of it caller don't wait for the callee and isn't blocked. We proposed the concurrency metric for synchronous method as follow:

,where $NoAsyncPath_{ikjl}$ denotes the set of path from $c_i.m_k$

$$mpc_{ikjl} = 1 - \frac{\left(\sum_{\forall c_x.m_y:mr_{xyik}=1} tmif_{xyik}\right) \times \sum_{\forall p \in NoAsyncPath_{ikjl}} \left(\Pi_{path}(p)\right)}{\sum_{\forall c_x.m_y:mr_{xyjl}=1} tmif_{xyjl}}$$

to $c_j.m_l$ in MDG, which includes only synchronous type of method as node except for start node, $c_i.m_k$ and end node, $c_j.m_l$. $\Pi_{path}(p)$ denotes the product of dmif of all edges included in path p.

3.2 Class Dependence Graph (CDG)

CDG is undirected and weighted graph, representing the communication patterns and several features of each class and interactions among classes. Each class in application is modeled as a node and the communication between two classes is modeled as an undirected edge. The weight of node denotes the total workloads of the class. The weight of arc indicates the communication costs and degree of concurrency between two classes. This graph has following components that are constructed from MDG:

$$\mathbf{CDG = (C, CR, CW, CCW, CPC)}$$

- **C** (Classes) = {c_i}, where c_i indicates class i.
- **CR** (Class Relationships) = {cr_{ij}}, where cr_{ij} = 1 if some method in class i invokes some method in class j. Otherwise cr_{ij} = 0. If i = j, c_{ij} = 0.
- **CW** (Class Workloads) = {cw_i}, where cw_i indicates the sum of the size of each method of class i in the number of instructions executed in run-time. It is calculated by

$$cw_i = \sum_{\forall \text{method } k \in \text{class } i} \; \sum_{\forall j, l : mr_{jlik} = 1} (mw_{ik} \times tmif_{jlik})$$

- **CCW** (Class Communication Workloads) = {ccw_{ij}}, where ccw_{ij} indicates the estimated communication cost between class i and class j (i ≠ j). If i = j, ccw_{ij}=0.

$$ccw_{ij} = \sum_{\forall \text{method } l \in \text{ class } j} \; \sum_{\forall \text{method } k \in \text{ class } i : \text{mr}_{ikjl} = 1} \left(miw_{ikjl} \times tmif_{ikjl} \right) + \sum_{\forall \text{method } k \in \text{ class } i} \; \sum_{\forall \text{method } l \in \text{ class } j : \text{mr}_{jlik} = 1} \left(miw_{jlik} \times tmif_{jlik} \right)$$

- **CPC** (Class Potential Concurrency) = {cpc_{ij}}, where cpc_{ij} denotes the potential parallelism between c_i and c_j (i ≠ j). This matrix is symmetric. It is calculated as follows:

$$cpc_{ij} = \sum_{\forall \text{method } l \in \text{ class } j : mit_{jl} = 1} \; \sum_{\forall \text{method } k \in \text{ class } i : \text{mr}_{ikjl} = 1} mpc_{ikjl} \times tmif_{ikjl} + \sum_{\forall \text{method } k \in \text{ class } i : mit_{ik} = 1} \; \sum_{\forall \text{method } l \in \text{ class } j : \text{mr}_{jlik} = 1} mpc_{jlik} \times tmif_{jlik}$$
$$+ \sum_{\forall \text{method } l \in \text{ class } j : mit_{jl} = 0} \; \sum_{\forall \text{method } k \in \text{ class } i : \text{mr}_{ikjl} = 1} mpc_{ikjl} + \sum_{\forall \text{method } k \in \text{ class } i : mit_{ik} = 0} \; \sum_{\forall \text{method } l \in \text{ class } j : \text{mr}_{jlik} = 1} mpc_{jlik}$$

The reason why we don't multiply mpc by tmif in case of mit = 0 (synchronous call) is that the influence of asynchronous call on the concurrency and performance is much more than that of synchronous call.

3.3 System Graph (SG)

System Graph (SG) is undirected and weighted graph, representing the attributes of each machine site and interconnection among computer or processors, such as computing power, network bandwidth, and connectivity among computers. Each computer or processor in distributed environment is modeled as a node and the network connection is modeled by an undirected edge. The weight of node denotes the computing power and the weight of arc denotes the network bandwidth of the connection. System Graph has following components:

$$\mathbf{SG = (P, PR, PC, NC)}$$

- **P** (Processors) = {p_i}, where p_i indicates the processor i in heterogeneous distributed environments.
- **PR** (Processor Relationships) = {pr_{ij}}, where pr_{ij} = 1 if processor i is network-connected with processor j. Otherwise pr_{ij} = 0.
- **PC** (Processor Capability) = {pc_i}, where pc_i indicates the average time of execution for one instruction on processor i.
- **NC** (Network Capability) = {nc_{ij}}, where nc_{ij} indicates the average time needed to transfer a unit of information from processor i to processor j.

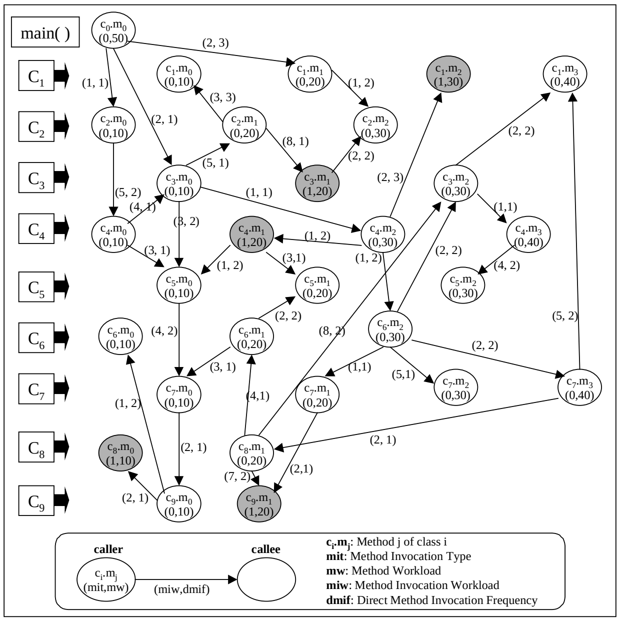


Figure 4. An example of MDG from C++ program

4. Experiments and Analysis

To evaluate our model, we implemented a partitioning and allocation method for C++ program using genetic algorithm (GA). Because this problem is multi-objective optimization with non-commensurable objectives (i.e. concurrency, communication cost and load balance), we adopt the Niched Pareto GA (NPGA) [7]. NPGA produces not a single solution, but a family of solutions known as the Pareto-optimal set, of which user can select optimal solution appropriate for his conditions. Each element in Pareto-optimal set is called non-dominated solution meaning that there are no other solutions superior in terms of all objectives.

Table 1. Matrix representing a CDG

	main	C1	C2	C3	C4	C5	C6	C7	C8	C9	CW
main	0 / 0 /1	1 / 6 /1	1/ 1 / 1	1/ 2 / 0.67	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	50
C1	1/ 6 / 1	0 / 0 / 1	1 / 33 / 0.5	1 / 144 /1	1 / 18 / 9	0 / 0 / 1	0 / 0 / 0.77	1 / 120 / 0.75	0 / 0 / 1	0 / 0 / 1	4260
C2	1 / 1 / 1	1 / 33 / 0.5	0 / 0 / 1	0 / 51 / 3.5	1 / 10 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	430
C3	1 / 2 / 0.67	1 / 144 /1	0 / 51 / 3.5	0 / 0 / 1	0 / 47 / 0.33	0 / 18 / 0.7	0 / 24 / 0.67	0 / 0 / 1	1 / 192 / 0.33	0 / 0 / 1	1170
C4	0 / 0 / 1	1 / 18 / 9	1 / 10 / 1	0 / 47 / 0.33	0 / 0 / 1	0 / 324 / 2.1	1 / 6 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	1670
C5	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 18 / 0.7	0 / 324 / 2.1	0 / 0 / 1	0 / 48 / 0.2	1 / 160 / 0.23	0 / 0 / 1	0 / 0 / 1	2960
C6	0 / 0 / 1	0 / 0 / 0.77	0 / 0 / 1	0 / 24 / 0.67	1 / 6 / 1	0 / 48 / 0.2	0 / 0 / 1	0 / 96 / 1	1 / 48 / 1	1 / 104 / 1	1460
C7	0 / 0 / 1	1 / 120 / 0.75	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	1 / 160 / 0.23	0 / 96 / 1	0 / 0 / 1	0 / 24 / 1	0 / 116 / 6	1300
C8	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	1 / 192 / 0.33	0 / 0 / 1	0 / 0 / 1	1 / 48 / 1	0 / 24 / 1	0 / 0 / 1	0 / 272 / 76	760
C9	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	0 / 0 / 1	1 / 104 / 1	0 / 116 / 6	0 / 272 / 76	0 / 0 / 1	1120

(cr_{ij} / ccw_{ij} / cpc_{ij})

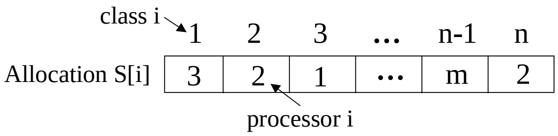


Figure 5. Representation of the Solution Structure

For example, a MDG in Figure 4 from some C++ application is considered. A grayed circle represents an asynchronous type of method. From this MDG, a CDG (in Table 1) is constructed by the mechanism explained in section 3. SG in our experiment consists of fully connected four nodes with the distinct computing power and network bandwidth.

In our GA implementation, the solution (chromosome) structure is an n-dimensional vector of integer alphabet in the set {1,…,m}. These integer alphabets identify the physical nodes of SG. The chromosome structure is illustrated in Figure 5.

The cost function is a vector of normalized fitness function component associated with each of objectives that are adapted to the minimization problem. Each objective function is as follows.

- f_1 **(Network Communication Cost)** : This is a objective function component that calculates the total amount of network communication cost of a solution by the following formula:

$$f_1 = \sum_{p} \sum_{q \neq p} \sum_{i} \sum_{j \neq i} ccw_{ij} \times nc_{pq} \times x_{ip} x_{jq}$$

where $x_{ip} = 1$ if class i is allocated to node p, otherwise $x_{ip} = 0$.

- f_2 **(Loss of Concurrency)** : This is a objective function component that calculates the total loss of concurrency of a solution by the following formula:

$$f_2 = \sum_{p} \sum_{i} \sum_{j \neq i} cpc_{ij} \times x_{ip} \times x_{jp}$$

- f_3 **(Load Imbalance)** : This is an objective function component that measures how the workloads of each class in one solution aren't distributed uniformly. It is calculated by the following formula:

$$f_3 = \left\{ \sum_{p} \left(\sum_{i} cw_i \times pc_p \times x_{ip} - TL / N \right)^2 \right\} / N$$

$$TL = \sum_{p} \sum_{i} cw_i \times pc_p \times x_{ip}$$

In our experiment, a random initial population size was set to 100 and a standard one-point crossover was used as crossover GA operation. Because in NPGA the value of σ_{share} and the tournament size t_{dom} play a critical role, the best result should be selected by the extensive experiment changing the values of σ_{share} and t_{dom}. By the experiment, when $\sigma_{share} = 0.18$ and $t_{dom} = 7$, our NPGA produces the best result. Initial population graph is illustrated in Figure 6, where three objectives, the communication cost, concurrency loss and load imbalance, represent x, y and z-axis respectively. The value of each objective function is normalized between 0 and 1, and it approaches to optimal solution when its value becomes less. Distribution of best solutions of our NPGA at generation 50000 is represented in Figure 7, where we can see that there are more non-dominated solutions (the solid line) than in the initial population. Three pairs of two-dimensional graph in Figure 8 demonstrate the distinction between the distribution of the initial and final population more prominently, where we can identify tradeoffs between the communication and load balance, and between concurrency and communication. Concurrency and load balance have no tradeoff each other.

Figure 6. Initial Population Graph in 3-Dimension

Figure 7. Final Population Graph at generation 50000

Figure 8. Three pairs of two-dimensional graph of Initial and Final Population

Figure 8. Three pairs of two-dimensional graph of Initial and Final Population (continued)

5. Conclusion

In this paper we have proposed a partitioning and allocation model for mapping OO application to heterogeneous distributed system. Our model is based on graph-theoretic approach, dealing with several characteristics of OO paradigm such as the inheritance, the dynamic binding, the method invocation frequency, the method invocation type. In addition, our approach treats a partitioning and allocation problem as multiobjective problem with non-commensurable objectives by defining three objective function components for load balance, degree of concurrency and communication cost individually. The Niched Pareto GA-based approach, techniques to experiment our model, finds a lot of possible solutions, of which users can select optimal allocation according to their environmental conditions. As well, it is demonstrated that there exist tradeoffs between concurrency and communication, and between communication cost and load balance, but concurrency and load balance have no tradeoff with each other.

Future works include the theoretic and practical comparison of our approach with other partitioning and allocation algorithms. Additionally, a model considering code migration, timing, fault tolerance and reliability should be defined.

References

[1] G. C. Low, G. Rasmussen and B. Henderson-Sellers, “Incorporation of distributed computing concerns into object-oriented methodologies”, *Journal of Object-Oriented Programming*, Vol. 9, No. 3, Jun. 1996.
[2] W. Chang, C. Tseng, “Clustering approach to grouping objects in message-passing systems”, *Journal of Object-Oriented Programming*, Vol 8, No. 6, Oct. 1995.
[3] U. Bellur, G. Craig, and D. Lea, “Clustering: Composition for Active Object Systems”, *In Proceedings of 27th Hawaii International Conference on System Science*, Jan. 1994.
[4] W. Chang, C. Tseng, “Supporting distributed objects in FIFO-based message-passing systems”, *Journal of Object-Oriented Programming*, Vol 7, No. 9, Feb. 1995.
[5] L. R. Welch, B. Ravindran, J. Henriques and D. K. Hammer, “Metrics and Techniques for Automatic Partitioning and Assignment of Object-based Concurrent Programs”, *In Proceedings of Seventh IEEE Symposium on Parallel and Distributed Processing*, Oct. 1995.
[6] P. Wegner and S. A. Smolka, “Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives,” *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 4, 1983, pp. 446-462.
[7] J. Horn and N. Nafpliotis, “Multiobjective Optimization Using The Niched Pareto Genetic Algorithm”, Technical Report No.93005, University of Illinois at Urbana-Champaign, USA, Jul. 1993.
[8] A. Stoyenko, J. Bosch, M. Aksit and T. Marlowe, “Load Balanced Mapping of Distributed Objects to Minimze Network Communication”, *Journal Of Parallel and Distributed Computing,* No. 34, pp. 117-136, 1996.
[9] S. Krishnan and L. Kale, “Automating Parallel Runtime Optimizations Using Post-Mortem Analysis”, *In Proceedings of 10th ACM International Conference on Supercomputing* , May 1996.
[10] C. Houstis, “Module Allocation of Real-Time Applications to Distributed Systems”, IEEE Transactions On Software Engineering, Vol. 16, No. 7, Jul. 1990.
[11] S. S. Stubbs, and D. L. Carver, “IPCC++: InterProcess Communication with C++”, *Journal of Object-Oriented Programming*, Feb. 1996.
[12] J. Aguilar, Heuristics to Optimize the Speed-up of Parallel Programs, *In Proceedings of 3rd International ACPC Conference with Special Emphasis on Parallel Database and Parallel I/O* , pp.174-83.
[13] D. E. Goldberg, “Genetic Algorithms in Search, Optimization, & Machine Learning”, Addison-Wesley, 1989.
[14] C. M. Fonseca and P. J. Fleming, “Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization”, In Proceedings of 5th International Conference On Genetic Algorithms, Jul. 1993.