A Partitioning Approach for Object-Oriented Software Development for Parallel Processing Systems

Stephen S. Yau, Doo-Hwan Bae and Gilda Pour
Computer and Information Sciences Department
University of Florida
Gainesville, Florida 32611-2024

Abstract

Existing partitioning approaches for distributed and parallel processing systems are not suitable for partitioning in object-oriented software development for parallel processing systems mainly due to the lack of shared data concept. In this paper, a partitioning approach for object-oriented software development for parallel processing systems is presented. The objective of our partitioning approach is to improve the overall performance of a software system by minimizing communication cost among processors and exploiting potential parallelism among objects. The software system is modeled by a graph, and a bottom-up clustering technique is presented to partition the objects into a set of clusters to achieve our goal.

1 Introduction

One of the important issues in the software development for parallel processing systems is the distribution of software components or modules to the processors so that the execution of the software system can be completed with a minimum amount of time. This process can be divided into two phases: *partitioning* and then *allocation*. In partitioning phase, the software system is partitioned into a set of modules. In allocation phase, the modules are assigned to the processors. Intuitively, to exploit parallel processing power, the modules should be distributed to as many processors as possible to be executed in parallel. On the other hand, to reduce high communication overhead among processors, the modules should be distributed over as few processors as possible. The tradeoff between these two conflicting criteria has been well known in parallel processing systems design. It is very difficult to increase the performance of the system in proportion to the number of processors in parallel processing systems due to communication costs between processors, contention of shared resources and inability to keep all the processors busy all the time[1]. This is one of the reasons for having a large gap between the ideal peak performance and the real performance in most parallel processing systems.

*The work was supported by the Rome Laboratory, U. S. Air Force Systems Command under contract No. F30602-91-C-0045.

The problem of partitioning for software development for distributed and parallel processing systems has been studied extensively. Existing approaches can be classified in three categories: graph-theoretic [2, 3], 0-1 integer programming [4] and heuristic[5]-[11]. Some approaches adopt more than one method. Partitioning approaches attempt to minimize the sum of communication time and execution time. Because this problem is NP-hard, a suboptimal solution is usually sought using heuristics.

Object-oriented software development has a number of advantages, especially maintainability and extensibility. In object-oriented software development for parallel processing systems [12], the software system is considered as a set of objects where every object can contain shared data that may be accessed by a number of objects. If the shared data is modified, the access must be serialized. When shared data is not modified, parallel invocation of methods in the object should be allowed. The way the objects are assigned to the processors can significantly affect the overall system performance. The existing approaches are generally not suitable for object-oriented software development for parallel processing systems because they do not consider the shared data concept.

In software development for parallel processing systems [13], partitioning phase precedes coding and allocation phases. Most existing approaches cannot be applied prior to coding and allocation because they require the information on the execution time for each module and the communication time among modules to be a part of their input while this information is unlikely available prior to coding and allocation phase. In this paper, we will present a partitioning approach for object-oriented software development for parallel processing systems. The objective of our partitioning approach is to improve the overall performance of the software system by minimizing communication cost among processors while maintaining the potential parallelism among objects. Our partitioning approach can be applied as early as at the end of object design phase. We will assume that object invocations are synchronous, and assignment to the processors is static.

2 Our Partitioning Approach

The behavior of objects in a software system can be identified as *parallel*, *sequential*, *selective*, or *waiting*. The input to our partitioning approach is the information available after the object design phase, and it consists of the object behavior, object invocation frequency, the upper limit on data units transferred between two objects at every invocation, and the number of replications of every object. Our partitioning approach has three stages: initialization, normalization, and clustering.

2.1 Initialization

The software system is modeled by an undirected weighted graph. The initial graph $G = (V, E)$ has a set of nodes V and a set of edges E such that:

- An object o_i is represented by node i in V.
- An edge (i, j) is in E if and only if o_i and o_j can communicate with one another or both objects can be invoked concurrently by another object.
- A node i has a non-negative weight, denoted by r_i, with a value equal to the number of replications of object o_i.
- An edge (i, j) has an ordered set of weights (u_{ij}, v_{ij}) where u_{ij} and v_{ij} are the communication and concurrency weights for that edge, respectively.

Communication and concurrency weights are assigned according to the following five rules. A communication weight is given a negative sign to imply the communication cost resulted from assigning the objects represented by the nodes connected by the edge to different processors. A concurrency weight is given a positive sign to imply the gain achieved as a result of parallel execution of the objects represented by the nodes connected by the edge to different processors. In this discussion, f_{ij} is the frequency of invocation between o_i and o_j, and d_{ij} is the upper limit on data units transferred between o_i and o_j at every invocation.

Rule 1. o_1 : CON$(o_2, o_3, \ldots, o_n)$ describes a case where $o_2, o_3, \ldots, o_{n-1}$, and o_n can be executed concurrently after being invoked by o_1. It corresponds to a subgraph $G = (V, E)$ where $V = \{i \mid 1 \leq i \leq n\}$ and $E = \{(i, j) \mid 1 \leq i < j \leq n\}$. Communication and concurrency weights are assigned to the edges in E as follows:

1) For $2 \leq i < j \leq n$, there are two possibilities:

  a) If (i, j) does not exist, then create (i, j) with weights $u_{ij} = 0$ and $v_{ij} = f_{1i} = f_{1j}$.

  b) If (i, j) exists, then u_{ij} remains unchanged, and $v_{ij} = v_{ij} + f_{1i}$.

2) For i = 1 and $2 \leq j \leq n$, there are two possibilities:

  a) If (i, j) does not exist, then create $(1, j)$ with weights $u_{ij} = -(f_{1j} \times d_{1j})$ and $v_{ij} = 0$.

  b) If (i, j) exists, then $u_{ij} = u_{ij} - (f_{1j} \times d_{1j})$ and v_{ij} remains unchanged.

Rule 2. o_1 : SEQ$(o_2, o_3, \ldots, o_n)$ describes a case where o_1 invokes o_j's for $2 \leq j \leq n$ in the sequential order $o_2, o_3, \ldots, o_n$. It corresponds to a subgraph $G = (V, E)$ where $V = \{i \mid 1 \leq i \leq n\}$ and $E = \{(1, j) \mid 2 \leq j \leq n\}$. In assigning communication and concurrency weights to the edges in E, there are two possible cases for $2 \leq j \leq n$:

1) If $(1, j)$ does not exist, then create $(1, j)$ with weights $u_{1j} = -(f_{1j} \times d_{1j})$ and $v_{1j} = 0$.

2) If $(1, j)$ exists, then $u_{1j} = u_{1j} - (f_{1j} \times d_{1j})$ and v_{1j} remains unchanged.

Rule 3. o_1 : ONE-OF($o_2, o_3, \ldots, o_n$) or o_1 : SEL($o_2, o_3, \ldots, o_n$) each describes a case where o_1 invokes only one of o_j's for $2 \leq j \leq n$. SEL is used when the selection depends on the True/False status of a boolean condition, but ONE-OF is used when the selection is done without checking any condition. The corresponding subgraph for either one is $G = (V, E)$ where $V = \{i \mid 1 \leq i \leq n\}$ and $E = \{(1, j) \mid 2 \leq j \leq n\}$. In assigning communication and concurrency weights to the edges in E, there are two possibilities for $2 \leq j \leq n$:

1) If $(1, j)$ does not exist, then create $(1, j)$ with weights $u_{1j} = -(f_{1j} \times d_{1j})/(n-1)$ and $v_{1j} = 0$.

2) If $(1, j)$ exists, then $u_{1j} = u_{1j} - (f_{1j} \times d_{1j})/(n-1)$ and v_{1j} remains unchanged.

Rule 4. o_i : WAIT (o_j) describes a case where o_i waits to be invoked by o_j. It corresponds to a subgraph $G = (V, E)$ where $V = \{i, j\}$ and $E = \{(i, j)\}$. There are two possibilities:

1) If (i, j) does not exist, then create $(1, j)$ with weights $u_{ij} = v_{ij} = 0$.[1]

2) If (i, j) exists, then both u_{ij} and v_{ij} remain unchanged.

Rule 5 is applied to the cases of nested clauses. Before presenting Rule 5, we define the preservation of the edge relationship, denoted by E-R, between two subgraphs. Let $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$ be two subgraphs where $V_A = \{x_1, \ldots, x_q\}$ and $V_B = \{y_1, \ldots, y_p\}$ for some $q \geq 1$ and $p \geq 1$. For every x in V_A and every y in V_B, one of the following holds:
E-R(x, y) = **true**, if edge (x, y) exists.
E-R(x, y) = **false**, otherwise.
Then the preservation of the edge relationship E-R

[1] A nonzero communication weight will be assigned to this edge when object o_j is processed.

between the two subgraphs is defined as follows: E-R(G_A, G_B) = $\bigwedge_{1\le i\le q, 1\le j\le p}$ E-R(x_1, y_1)

Rule 5. It is applied when nested clauses are used to specify the object behavior. The steps are:

1) Modifying the object behavior by substituting every nested clauses with one dummy object.

2) For every dummy object introduced in step 1:

2.1) Applying the appropriate rule(s) and preserving its edge relationships with other objects.

2.2) Assigning communication and concurrency weights to edges using Rules 1-4.

2.2 Normalization

As stated earlier, the goal of our partitioning approach is to make a trade-off between two conflicting criteria of minimizing communication cost between the processors and exploiting potential parallelism among the objects. In other words, it is desirable to find an optimal point at which communication costs are reasonably reduced while the parallel execution of objects is well achieved. However, finding an optimal solution requires execution and communication time to be available which is unlikely prior to coding and allocation. Even if such information were available, the problem of clustering to be discussed in the next section is NP-hard. As a result, our approach would provide a suboptimum solution.

In order to accommodate the two conflicting partitioning subgoals, we present a normalization method so that the communication and concurrency weights associated to every edge can be combined to obtain a common metric for the two kinds of weights. Let u_{min} be the minimum communication weight and v_{max} be the maximum concurrency weight. First for every edge (i, j), we replace u_{ij} with $-u_{ij}/u_{min}$ and v_{ij} with v_{ij}/v_{max}. This brings all communication weights to the range of (-1, 0) and all concurrency weights to the range of (0, 1). Then, we define a new edge weight w_{ij}, called *gain*, to replace (u_{ij}, v_{ij}). The value of w_{ij} is taken to be $\alpha \times u_{ij} + (1 - \alpha) \times v_{ij}$ where α is in the range of (0, 1). To obtain a suitable α, the exact figure of the parallel machine, the exact execution time and the exact communication cost are needed. Our partitioning approach is applied before coding and allocation phases where this information is unlikely to be available. Therefore, modification of α is allowed in the sense that if after allocation, the results of partitioning turns out not to be satisfactory, the overall performance of the software system can be tuned by using the required information available at the end of allocation phase to find a suitable α and then repeating the last two stages of our approach for the new α. Next, we define the objective function Y to be sum of real-value weights of all edges in the graph.

2.3 Clustering

The main objective of this stage is to maximize the value of Y by taking a bottom-up approach to cluster the objects represented by the nodes connected by the edges with negative weight values. Note that an edge with a positive weight suggests that parallel execution of the objects represented by the nodes connected by the edge will reduce the execution time of the software system. Hence, these objects should not be in one cluster. On the other hand, an edge with a negative weight suggests that the objects represented by the nodes connected by the edge should be placed in the same cluster because execution of these objects on different processors will increase the communication cost among processors. If the weight of an edge is equal to zero, we choose not to cluster the objects represented by the nodes of that edge together because clustering of such objects does not increase the value of Y. Furthermore, comparing a partition consisting of many small processes with one consisting of a few large processes, the partition with many small processes will provide the allocation phase with more flexibility for the purpose of load balance or growth potential.

The input to clustering is an undirected weighted graph $G' = (V', E')$ where $V = V'$, $E = E'$, and $G = (V, E)$ is the initial graph. The difference between G and G' is that every edge $(i, j) \in E$ has a set of weights (u_{ij}, v_{ij}) while the same edge $\in E'$ has a weight w_{ij} representing the degree of contribution to improving the overall system performance that is made by the parallel execution of the objects represented by the nodes connected by that edge.

We define function SIZE to map every node in the graph to a positive integer that is equal to the number of objects in the cluster represented by that node. The steps of clustering stage are listed below.

1. **for** every node c **do** Set SIZE(c) = 1.
while there is an edge with a negative weight **and** there is more than one node in the graph **do**
begin
2. Find edge (a, b) where w_{ab} is not greater than any other edge weight in the graph.
3. Group a and b to form a new cluster q.
Set SIZE(q) = SIZE(a) + SIZE(b).
4. **for** every node c there are four possible cases:
C.1 **if** E-R(c, a) = **true and** E-R(c, b) = **true then**
E-R(c, q) = **true** .
Assign weight $(w_{ca} + w_{cb})$ to edge (c, q) .
C.2 **if** E-R(c, a) = **true and** E-R(c, b) = **false then**
E-R(c, q) = **true**.
Assign weight w_{ca} to edge (c, q).
C.3 **if** E-R(c, a) = **false and** E-R(c, b) = **true then**
E-R(c, q) = **true**.
Assign weight w_{cb} to edge (c, q).
C.4 **if** E-R(c, a) = **false and** E-R(c, b) = **false**

then
E-R(c, q) = **false**.

5. **if** (SIZE(a) = 1 **and** $r_a \geq 1$) **then**
Add edge (a, q) to the graph.
Assign $+\infty$ to edge (a, q).
Set $r_a = r_a$ - 1.
else
Delete node a.
for every node c **do**
if E-R(c, a) = **true then**
Assign E-R(c, a) = **false**.

6. Repeat Step 5 for node b.
end

7. **if** all replicated objects are not in the graph **then**
for every node c **do**
if (SIZE(c) = 1 **and** $r_c = k \geq 1$) **then**
Add k new nodes each to represent one replication of object o_c.
Obtain the edges incident to a new node by duplicating the edges incident to c and their weights.
Connect any two of new nodes to one another and also any one of them to node c and assign a weight $+\infty$ to any new edge.
Go to while statement.

In Step 1, the value of function SIZE at every node in the graph is set to 1 because a node in the initial graph represents only one object. Steps 2-6 are executed until there is no edge with a negative weight or there is only one node in the graph. In Step 2, edge (a, b) with the minimum weight w_{ab} is found. In Step 3, nodes a and b are clustered to form new node q. The value of function SIZE at node q is the sum of the number of objects in nodes a and b. Forming node q calls for addition and deletion of some edges along with modification of their weights. This process is done in Step 4. Node a is deleted along with all edges incident to it unless r_a has a positive value which means that there are at least r_a replications of object o_a that have not been yet represented in the graph. If node a is not deleted, then it must be connected to node q, and the weight $+\infty$ has to be assigned to edge (a, q). r_a is decremented by 1 in order to be equal to the number of the replications of o_a that have not yet been represented in the graph. All this is done in Step 5. Step 6 is the repetition of Step 5 for node b. The **while** loop runs until there is no more edge with a negative weight or there is only one node in the graph. In Step 7, any node c representing only object o_c, but not its replications, is identified, and r_c new nodes are added to the graph each representing one replication of o_c. The set of edges incident to any of these new nodes is duplication of the set of edges incident to node c. The weight of any one of these edges is the same as the weight of its counterpart. In order to ensure that the replications of an object do not end up in one cluster, the nodes representing any two such objects are connected by an edge with weight $+\infty$. If any node is added to the graph in Step 7, the **while** loop is repeated until there is no edge with a negative weight in the graph or there is only one node.

The output of clustering is an undirected weighted graph in which every node q represents a cluster of object(s) and every edge (p, q) has a positive weight representing the degree of contribution to improving the overall system performance that can be made by parallel execution of clusters p and q. Note that a larger weight of an edge implies that more gain in improving the overall performance of the software system can be obtained as a result of allocating two clusters of objects represented by the nodes connected by the edge on two different processors.

3 Time Complexity

Let m be the number of objects in the software system, n be the number of objects without considering replications of the objects, and e be the total number of edges if all objects including replicated ones were represented in the initial graph. Step 1 takes O(n) time to run. Step 2 runs at most in O(e) time. Step 3 has a constant running time. Step 4 runs in O(n) time. In Step 5, the time complexity of **else** part dominates that of **if-then** part because **else** part takes O(n) and **if-then** part takes constant time. Step 6 is simply the repetition of Step 5. The **while** loop will be executed at most O(**min**(e, m)) time. Hence, the entire loop runs at most in O(**min**(e, m) $\times$ **max**(e, n)) time. When there is at least one replicated object in the graph, e cannot be smaller than n. The reason is that a node i in the initial graph represents object o_i which has some relationship with at least another object o_j and this relationship requires edge (i, j) to be in the initial graph. Because e cannot be smaller than n, **max**(e, n) is always equal to e. With similar reasoning, we can show that **min**(e, m) is equal to m. Therefore, the entire loop can be executed at most in O(em) time. Step 7 can also run in O(em) time. Thus, the clustering algorithm can be completed in no more than O(em) time.

4 An Example

In order to illustrate our partitioning approach, we apply it to a Warehouse Management System(**WMS**). A brief statement of the requirements of the **WMS** is as follows:

> The warehouse management system interacts with manufacturers and customers such as retailers. Manufacturers generate items and send them to the warehouse manager and items are stored on the warehouse racks. The warehouse manager retrieves items from warehouse-racks and sends them to the customers upon their requests. The capacity of this warehouse is fixed. Reports of transaction information are generated periodically.

The object-oriented model of **WMS** consists of the following objects: o_1 = report-generator, o_2 = rack, o_3 = transaction, o_4 = product-information, o_5 = customer-server, o_6 = purchaser, and o_7 = check-out-counter.

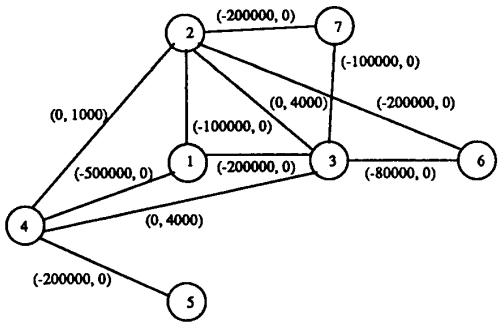

Figure 1: Initial Graph

1) Object behavior of the system can be described as

$o_1 : CON(o_2, o_3, o_4)$
$o_5 : SEQ(o_4)$
$o_6 : CON(o_2, o_3)$
$o_7 : CON(o_2, o_3)$

2) The frequencies of the object invocations and the upper limit on data units transferred between two objects every time one invokes the other (i.e. the two objects communicate) are assumed to be as follow. Those not listed are equal to zero.

$f_{12} = 1000$ $d_{12} = 100$
$f_{13} = 1000$ $d_{13} = 200$
$f_{14} = 1000$ $d_{14} = 500$
$f_{26} = 2000$ $d_{26} = 100$
$f_{27} = 1000$ $d_{27} = 200$
$f_{36} = 2000$ $d_{36} = 40$
$f_{37} = 1000$ $d_{37} = 100$
$f_{45} = 2000$ $d_{45} = 100$

3) Let r_i be the number of replications of o_i, $i = 1, 2, \ldots, 7$.

$r_1 = 1$ $r_2 = 0$ $r_3 = 0$ $r_4 = 0$
$r_5 = 0$ $r_6 = 0$ $r_7 = 0$

The three stages of partitioning are as follows:

1. Initialization Figure 1 shows the initial graph where o_i is modeled by node i, and every edge (i, j) in the graph has weights (u_{ij}, v_{ij}) where u_{ij} is the communication weight and v_{ij} is the concurrency weight associated with (i, j).

2. Normalization Figure 2 shows the graph after u_{ij} and v_{ij} are normalized and combined into a new weight w_{ij}, called gain, assuming α to be 0.5.

3. Clustering The edge connected by nodes 1 and 4 has a minimum weight with a negative value. Hence the two nodes should be clustered together. Because r_1 is 1 (i.e. object o_1 has a replication), node 1 should remain in the graph. A new edge is added to the graph which connects node 1 and the newly formed node (1,4). In order to ensure that o_1 and its replication are not placed in the same cluster, a weight $+\infty$ is assigned to this new edge. The resulting graph is shown in Figure 3.

The edge connecting node 5 and the node (1,4) has a minimum weight with a negative value. This suggests that the two nodes should be clustered. Consequently, o_1, o_4, and o_5 are placed in one cluster as shown in Figure 4.

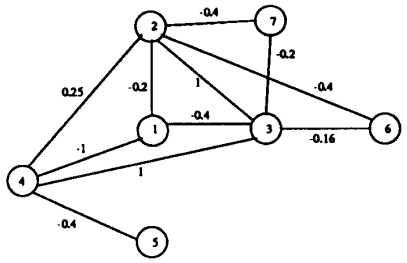

Figure 2: Graph at the End of Normalization Stage.

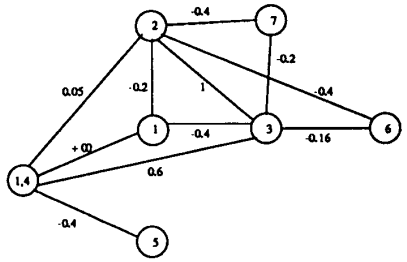

Figure 3: Graph at the End of First Clustering.

The edge connecting nodes 1 and 3 has a minimum weight with a negative value. Hence these nodes should be clustered as shown in Figure 5.

Next, the edge connecting nodes 2 and 7 has a minimum weight with a negative value which suggests that these nodes should be clustered as shown in Figure 6.

The edge connecting node 6 and the node (2,7) has a minimum weight with a negative value. Hence the two nodes should be clustered together. As a result, o_2, o_6 and o_7 are placed in one cluster. The graph after creating this new cluster is shown in Figure 7 in which there is no edge with a negative weight. Therefore, clustering stops, and the graph in Figure 7 is the output of partitioning phase.

5 Discussion

In this paper, we have presented a partitioning approach for object-oriented software development for parallel processing systems which incorporates the shared data concept [12] with synchronous access. The objective of our partitioning approach is to improve the overall performance of the software system through a trade-off of minimizing communication cost among processors and exploiting the potential parallelism among objects. The output of our partitioning approach is an undirected weighted graph in which every node represents a cluster of object(s) and every edge has a weight. A larger weight of an edge implies that parallel execution of the clusters represented by two nodes connected by the edge can improve the overall system performance to a larger extent. Hence at the allocation phase, if the number of processors available is smaller than the number of nodes in the output graph of our partitioning approach, the nodes incident

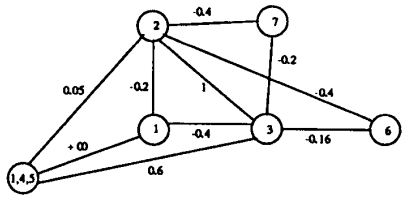

Figure 4: Graph at the End of Second Clustering.

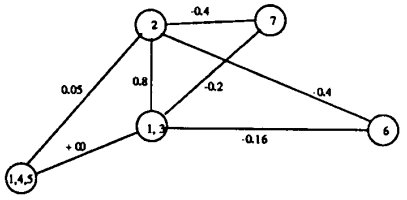

Figure 5: Graph at the End of Third Clustering.

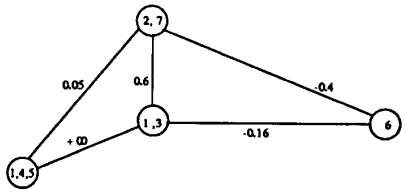

Figure 6: Graph at the End of Fourth Clustering.

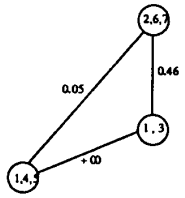

Figure 7: Output Graph.

to the edges with smaller weights are the candidates for clustering.

Currently, we are extending our partitioning approach through introducing new constraint imposed by limited processor memory size into the model. We are also conducting a sensitivity analysis on the normalization parameter. The preliminary results are encouraging.

References

[1] D. E. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Trans. on Computers*, Vol. 38, No. 3, 1989, pp. 408–423.

[2] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, 1977, pp. 85–93.

[3] C. C. Shen and W. T. Tsai, " A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computers*, Vol. 34, No. 3, 1985, pp. 197–203.

[4] O. I. El-Dessouki and W. H. Huan, "Distributed Enumeration on Network Computers," *IEEE Trans. on Computers*, Vol. C-29, No. 9, 1980, pp. 818–825.

[5] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, Vol. 15, No. 6, 1982, pp. 50–56.

[6] P. R. Ma and E. Y. S. Lee, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 1, 1982, pp. 41–47.

[7] S. M. Shatz and S. S. Yau, " A Partitioning Algorithm for Distributed Software Systems Design," *Information Sciences*, Vol. 38, No. 2, 1986, pp. 165-180.

[8] S. S. Yau and I. Wiharja, "An Approach to Module Distribution for the Design of Embedded Distributed Software Systems," *Information Sciences* , Vol. 56, No. 1, 1991, pp. 1-22.

[9] Virginia Mary Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. on Computers*, Vol. C-37, No. 11, 1988, pp. 1384-1397.

[10] H. Kasahara and N. Seinosuke, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, Vol. C-33, No. 11, 1984, pp. 1023–1029.

[11] V. Sarkar and V. Hennessy, "Partitioning Parallel Programs for Macro-dataflow," *Proc. ACM Conf. in Lisp and Functional Programming*, 1986, pp. 202–211.

[12] S. S. Yau, X. Jia, D.-H. Bae, M. Chidambaram, and G. Oh, "An Object-Oriented Approach to Software Development for Parallel Processing System," *Proc. 15th Annual Int'l Computer Software & Applications Conf. (COMPSAC91)*, October 1991, pp. 453-458.

[13] S. S. Yau, D.-H. Bae, and M. Chidambaram, "A Framework for Software Development for Distributed Parallel Computing Systems," *Proc. Third IEEE Workshop on Future Trends of Distributed Computing Systems*, April 1992, pp 240-246.