アルゴリズム89－2
（2003．3．14）

非同期共有メモリモデル上の資源割当て分散アルゴリズム

橋本健二*，　渡邊清高†，　濱田幸弘‡

概要

非同期 read-modify-write 型共有メモリモデル上で静的な資源割当て問題を解く分散アルゴリズムを提案する．ユーザが必要とする資源の明細に対し，対応する競合グラフが構成され，その頂点に頂点彩色アルゴリズムにより色がつけられていると仮定する．そのとき，提案するアルゴリズムは資源割当て問題を解き，アルゴリズムによる各ユーザの最大待ち時間は $O(k(m+k)l+kc)$ である．ここで，k は競合グラフの頂点彩色に用いられた色の数，m は同じ色をもつ頂点の最大数，l は各プロセスの連続するアクション間，ただし先行するアクションは共有変数を必要としない場合の時間の上界，c は任意のユーザが資源を利用する時間の上界である．

A Distributed Resource-Allocation Algorithm on the Asynchronous Shared Memory Model

Kenji Hashimoto*, Kiyotaka Watanabe†, and Yukihiro Hamada‡

Abstract

We propose a distributed algorithm for the static resource-allocation problem on the asynchronous read-modify-write shared memory model. We assume that for a resource-requirement specification for users, the corresponding conflict graph is constructed and that it is colored by a node coloring algorithm. Then, the proposed algorithm solves the resource-allocation problem and the maximum waiting time using the proposed algorithm is $O(k(m+k)l+kc)$, where k is the total number of colors used to color the nodes in the conflict graph, m is the maximum number of nodes that have the same color, l is an upper bound on the time between successive actions of each process where the preceding action does not involve any shared variables, and c is an upper bound on the time that any user spends using the resources.

1 Introduction

Resource allocation is the problem of managing a finite collection of distinct resources on the requests of users. We assume that every resource is indivisible that can only support one user at a time. We also assume that resources that each user needs to perform its work are known in advance and that they are not changeable. Each user may require an arbitrary but finite number of resources. Every

*大阪大学基礎工学部情報科学科
Department of Information and Computer Sciences, Osaka University
k-hasimt@exp.ics.es.osaka-u.ac.jp
†大阪大学基礎工学部システム科学科
Department of Systems Science, Osaka University
ks0710wk@ecs.cmc.osaka-u.ac.jp
‡明石工業高等専門学校電気情報工学科
Department of Electrical and Computer Engineering, Akashi National College of Technology
hamada@akashi.ac.jp

user can start its work only when it is granted to use all the resources that it needs. Thus, the resource-allocation problem is one that is generalized both from the mutual exclusion problem and from the dining philosophers problem. The drinking philosophers problem, introduced in [1], is one that is generalized from the resource-allocation problem.

Choy and Singh solved the resource-allocation problem on the asynchronous network model [2]. Although they considered the case where a resource is shared between two users, the algorithm given in [2] can be extended to the general case where resources may be shared by more than two users.

Lynch solved the resource-allocation problem on the asynchronous read-modify-write shared memory model [4], [5]. The algorithm stated in [5] requires pre-processing of the resource graph G for a given resource specification. The nodes of the resource graph G represent the resources, and there is an edge between two nodes exactly if there is some user that needs both associated resources. The algorithm in [5] uses $color(j)$ for each node j in G, where $color(j)$ is the color that is assigned to node j by a node coloring algorithm. Based on the node coloring, a total ordering of all the resources is defined in some way. Each read-modify-write shared variable in the shared memory model is associated with a resource, and the variable is used to manage that resource's queue. Each process in the shared memory model acts as an agent on behalf of the corresponding user. In the algorithm in [5], each process seeks its resources in increasing order one by one according to the total ordering of the resources. A process seeks a resource by putting its index at the tail of that resource's queue. The process obtains the resource when its index reaches the head of that resource's queue. After being granted all the resources, a process starts to perform its work. When a process completes its work, it returns all of its resources by removing its index from their queues. The maximum waiting time using the algorithm in [5] is $O(\mu^\kappa c + \kappa\mu^\kappa l')$, where κ is the total number of colors used to color the nodes in the resource graph, μ is the maximum number of processes that require any single resource, l' is an upper bound on the time between successive actions of each process, and c is an upper bound on the time that any user spends using the resources.

In this paper, we propose a distributed algorithm for the resource-allocation problem on the asynchronous read-modify-write shared memory model. We assume that for a resource-requirement specification for users, the corresponding conflict graph is constructed and that it is colored by a node coloring algorithm. Then, the proposed algorithm solves the resource-allocation problem and the maximum waiting time using the proposed algorithm is $O(k(m+k)l + kc)$, where k is the total number of colors used to color the nodes in the conflict graph, m is the maximum number of nodes that have the same color, l is an upper bound on the time between successive actions of each process where the preceding action does not involve any shared variables, and c is an upper bound on the time that any user spends using the resources. Although we cannot directly compare the maximum waiting time by the proposed algorithm with that of the algorithm in [5], the former is much less than the latter since the former is bounded by polynomials while the latter is bounded by exponentials.

2 Preliminaries

2.1 Asynchronous Shared Memory Model

We use a computation model called asynchronous shared memory model. It consists of a finite collection of processes $P_1, P_2, \cdots, P_n$ interacting with each other by means of a finite collection of shared variables. Each process is a simple type of state machine in which the transitions are associated with actions. The actions are classified as either input, output, or internal. We assume that each process have a port on which it can interact with its corresponding user using input and output actions. For each i, $1 \leq i \leq n$, process P_i corresponds to user U_i. Each user U_i is also modelled as a state machine. The input actions of each process are assumed not to be under the process's control; they are given by its corresponding user. The internal and output actions of each process are controlled by the process.

Each process performs its actions at varied speed. We depict users and the asynchronous shared memory model in Figure 1.

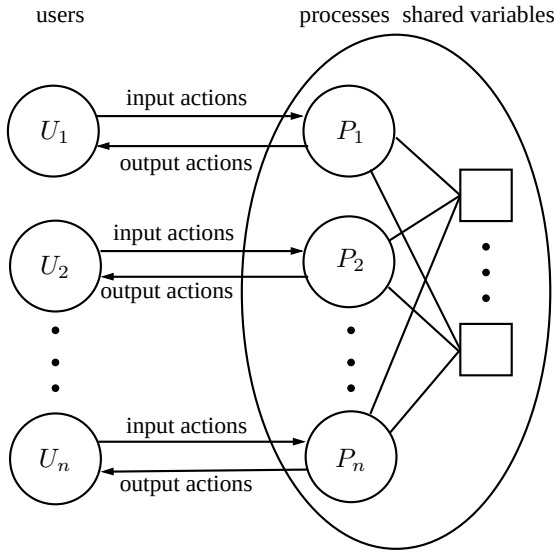

Figure 1: Users and asynchronous shared memory model.

Shared variables used here are of read-modify-write type. In one instantaneous read-modify-write operation on a shared variable x, a process P_i can do all of the following:

1. Read x.
2. Carry out some computation, possibly using the value of x, that modifies the state of P_i and determines a new value for x.
3. Write the new value to x.

The shared variables are used only for communication among the processes in the model. The shared memory model requires that each access to the variable should be indivisible, and that all processes should get fair turns to perform such accesses.

When we analyse the time complexity of algorithms on the asynchronous shared memory model, we make the following assumptions. There exists an upper bound l on the time between successive actions of each process where the preceding action does not involve any shared variables. The time between successive actions of each process where the preceding action involves a shared variable is at most dl if the variable is shared by exactly d processes.

2.2 Resource-Allocation Problem

Firstly we define a resource specification for n users $U_1, U_2, \cdots, U_n$. It consists of a finite set S of indivisible resources and n subsets $r_1, r_2, \cdots, r_n$ of S, where for every i, $1 \leq i \leq n$, the elements of r_i are resources that user U_i needs to perform its work. We say that two users U_i and U_j conflict with respect to a given resource specification if $r_i \cap r_j \neq \emptyset$.

When a user is not involved in any way with the resources, it is said to be in the remainder region R. Every user is in its remainder region in the initial state. When a user requires the resources, it executes a trying protocol. A user in such states is said to be in the trying region T. A user with access to the resources is said to be in the critical region C. After a user completes its work using the resources, it releases them. This is done by executing an exit protocol. A user in such states is said to be in the exit region E. Each user follows a cycle, moving from its remainder region R to its trying region T, then to its critical region C, then to its exit region E, and then back again to its remainder region R. We assume that there exists an upper bound c on the time that any user spends in C.

The shared memory model contains n processes $P_1, P_2, \cdots, P_n$, where for each i, $1 \leq i \leq n$, P_i corresponds to user U_i. The inputs to process P_i are try_i actions and $exit_i$ actions. The try_i action models a request by user U_i for access to the resources and the $exit_i$ action models an announcement by user U_i that it is done with the resources. The outputs of process P_i are $crit_i$ actions and rem_i actions. The $crit_i$ action models the granting of the resources to U_i and the rem_i action models a notification to U_i that the resources have been released. A sequence of try_i, $crit_i$, $exit_i$, and rem_i actions is said to be well-formed for user U_i if it is a prefix of the cyclically ordered sequence $try_i, crit_i, exit_i, rem_i, try_i, \cdots$.

The combination of an asynchronous shared memory model and the corresponding users is called asynchronous shared memory system. We now de-

scribe the conditions that the asynchronous shared memory system must satisfy in order to solve the resource-allocation problem.

Well-formedness: In any execution, and for any $i(1 \leq i \leq n)$, the subsequence describing the interaction between U_i and P_i is well-formed for i.

Exclusion: There is no reachable system state in which conflicting two users are in their critical regions.

Progress for the trying region: If at least one user is in T and no user is in C, then at some later point some user enters C.

Progress for the exit region: If at least one user is in E, then at some later point some user enters R.

Lockout-freedom for the trying region: If all users always return the resources, then any user that reaches T eventually enters C.

Lockout-freedom for the exit region: Any user that reaches E eventually enters R.

Note that the resource-allocation problem includes the mutual exclusion problem and the dining philosophers problem as special cases. Note also that we have stated the correctness conditions in terms of user regions. The process states are also classified according to their regions; these regions correspond exactly to the user regions. Thus we can equivalently state the correctness conditions in terms of process regions. We describe interchangeably user regions and process regions in the rest of this paper.

Here, we should mention a potentially serious issue. We tackle the resource-allocation problem on the asynchronous read-modify-write shared memory system. In any execution, there must be no reachable system state in which more than one process uses a read-modify-write shared variable at the same time. In other words, when process P_i tries to access some shared variable x, if x is used by the other process P_j, then P_i must wait until P_j releases x. This is very close to what is required for the resource-allocation problem. It almost seems as if we are assuming a solution to the very problem we are trying to solve.

One of possible answers to the issue is that several read-modify-write shared variables can be combined into a single multipart read-modify-write shared variable, and a single read-modify-write shared variable can be implemented by a combination of a single-writer/multi-reader shared memory system and a mutual exclusion algorithm. For the definition of the single-writer/multi-reader shared memory system and the mutual exclusion algorithm, please refer to [5].

2.3 Conflict Graph

For a given resource specification for n users $U_1, U_2, \cdots, U_n$, we define a conflict graph $G = (V, E)$ as follows:

$$\begin{aligned} V &= \{U_1, U_2, \cdots, U_n\}, \\ E &= \{(U_i, U_j) \mid U_i, U_j \in V, U_i \text{and} U_j \text{conflict}\}. \end{aligned}$$

For example, consider 4 users U_1, U_2, U_3, U_4 and the set $S = \{a, b, c, d\}$ of resources. Let $r_1 = \{a, b\}, r_2 = \{a, c\}, r_3 = \{b, d\}$ and $r_4 = \{c, d\}$, where r_i denotes the set of resources that U_i needs for each $i(1 \leq i \leq n)$. Then, the conflict graph for the given resource specification for 4 users is as shown in Figure 2. In this case, every edge in the conflict graph can be labelled with a singleton of resources. Generally, each edge in a conflict graph is labelled with a subset of resources.

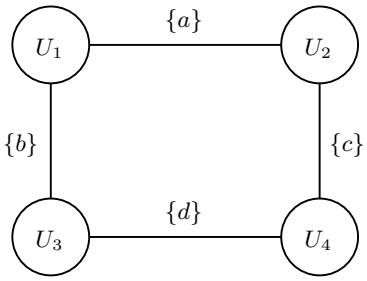

Figure 2: Conflict graph.

Color-grouping algorithm, described in Section 3.1, requires pre-processing of the conflict graph G for a given resource specification for n users. In particular, the *color-grouping algorithm* uses $color(i)$ for each process P_i, $1 \leq i \leq n$, where $color(i)$ is the color that is assigned to node U_i in G by the algorithm shown below. This algorithm colors nodes

in G with at most $\Delta + 1$ colors, where Δ is the maximum degree of G.

```
/* Initially all edges are unmarked
     and all nodes are uncolored */
for i := Δ downto 0 do
    while (∃v [v is not colored
          and has i unmarked edges]) do begin
        color(v) := i;
        mark all edges of v;
    end;
```

3 Color-Grouping Algorithm

3.1 Color-Grouping Algorithm

In this section, we introduce a distributed algorithm for the resource-allocation problem on the asynchronous shared memory system. We describe the algorithm in a precondition-effect style. This style groups together all the transitions that involve each particular type of action into a single piece of code. The code specifies the conditions under which the action is permitted to occur, and describes the changes that occur as a result of the action. The entire piece of code is assumed to executed indivisibly as a single transition.

From the definition of conflict graph, any two users that conflict are assigned different colors by a node coloring of the conflict graph. Thus, if two or more users in T have the same color, then they can enter C at the same time. This is the fundamental idea behind the *color-grouping algorithm* shown below. Processes in T that have the same color may form two groups, i.e., processes that entered T later must wait and comprise another group.

We assume that the conflict graph is constructed from a given resource specification for n users, and that it is colored by the node coloring algorithm described in Section 2.3. We also assume that each process knows which color is assigned to it. For each i, $1 \leq i \leq n$, $color(i)$ denotes the color that is assigned to process P_i. Let k denote the total number of colors used to color the nodes in the conflict graph. Let m denote the maximum number of processes that have the same color.

***Color-grouping algorithm*:**
Read-modify-write shared variables:
for every j, $1 \leq j \leq k$:
  $count(j) \in \{-m, \cdots, -1, 0, 1, \cdots, m\}$, initially 0,
  accessible by all processes that have color j
  $wait(j) \in \{0, 1, \cdots, m\}$, initially 0,
  accessible by all processes that have color j
$COLORQ$, a queue of color indices of length at most k, initially empty, accessible by all processes

Actions of P_i:

Input:	Internal:
try_i	$first\text{-}door_i$
$exit_i$	$reserve_i$
Output:	$test_i$
$crit_i$	$accept_i$
rem_i	$elect_i$
	$second\text{-}door_i$
	$keep_i$
	$inverse_i$
	$third\text{-}door_i$
	$disposal_i$
	$remove\text{-}queue_i$
	$reset_i$

States of P_i:
$pc \in \{rem, first\text{-}door, reserve, test, accept, elect, second\text{-}door, keep, inverse, third\text{-}door, leave\text{-}try, crit, disposal, remove\text{-}queue, reset, leave\text{-}exit\}$, initially rem
$num \in \{0, 1, \cdots, m\}$, initially 0

Transitions of P_i:

try_i
  Effect:
    $pc := first\text{-}door$

$first\text{-}door_i$
  Precondition:
    $pc = first\text{-}door$
  Effect:
    if $count(color(i)) \geq 0$ then
      $num := count(color(i))$
      $count(color(i)) := count(color(i)) + 1$
      $pc := elect$
    else
      $pc := reserve$

$reserve_i$
  Precondition:
    $pc = reserve$
  Effect:
    $wait(color(i)) := wait(color(i)) + 1$
    $pc := test$

$test_i$
  Precondition:
    $pc = test$
  Effect:
    if $count(color(i)) \geq 0$ then
      $num := count(color(i))$
      $count(color(i)) := count(color(i)) + 1$

$pc := accept$

$accept_i$
Precondition:
$pc = accept$
Effect:
$wait(color(i)) := wait(color(i)) - 1$
$pc := elect$

$elect_i$
Precondition:
$pc = elect$
Effect:
if $num = 0$ then
add $color(i)$ to $COLORQ$
$pc := second\text{-}door$
else
$pc := third\text{-}door$

$second\text{-}door_i$
Precondition:
$pc = second\text{-}door$
Effect:
if $color(i)$ is first on $COLORQ$ then
$pc := keep$

$keep_i$
Precondition:
$pc = keep$
Effect:
if $wait(color(i)) = 0$ then
$pc := inverse$

$inverse_i$
Precondition:
$pc = inverse$
Effect:
$count(color(i)) := -count(color(i))$
$pc := leave\text{-}try$

$third\text{-}door_i$
Precondition:
$pc = third\text{-}door$
Effect:
if $count(color(i)) < 0$ then
$pc := leave\text{-}try$

$crit_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

$exit_i$
Effect:
$pc := disposal$

$disposal_i$
Precondition:
$pc = disposal$
Effect:
if $count(color(i)) < -1$ then
$count(color(i)) := count(color(i)) + 1$
$pc := leave\text{-}exit$
else
$pc := remove\text{-}queue$

$remove\text{-}queue_i$
Precondition:
$pc = remove\text{-}queue$
Effect:
remove $color(i)$ from $COLORQ$
$pc := reset$

$reset_i$
Precondition:
$pc = reset$
Effect:
$count(color(i)) := 0$
$pc := leave\text{-}exit$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

3.2 Correctness and Time Complexity of the Algorithm

We first describe a correspondence between the region designations R, T, C, and E and the values of pc: R corresponds to rem; T corresponds to $first\text{-}door, elect, second\text{-}door, keep, inverse, leave\text{-}try, third\text{-}door, reserve, test$, and $accept$; C corresponds to $crit$; and E corresponds to $disposal, leave\text{-}exit, remove\text{-}queue$, and $reset$.

During the execution of the *color-grouping algorithm*, several processes in T that have the same color form a group (although the group is not formed explicitly), and they enter C almost simultaneously. Let G denote a group of processes in T and let $color(G)$ denote the color of group G. During the execution of the *color-grouping algorithm*, for each group G, one process in G, say P_{GT}, inserts $color(G)$ at the tail of queue $COLORQ$, and one process in G, say P_{GE}, deletes $color(G)$ from the head of $COLORQ$. Let pc_i denote pc of process

P_i. For each color j, $1 \leq j \leq k$, let $S_j = \{P_i \mid pc_i \in \{accept, elect, second\text{-}door, keep, inverse, third\text{-}door, leave\text{-}try, crit, disposal, remove\text{-}queue, reset\}$ and $color(P_i) = j\}$.

Lemma 1 *During the execution of the color-grouping algorithm, if $count(j) = 0$, then $S_j = \emptyset$ for every $j (1 \leq j \leq k)$.*

Proof Sketch: Let j be an arbitrary color. In the initial system state, apparently $count(j) = 0$ and $S_j = \emptyset$.

In some reachable system state s, we assume that $count(j) = 0$ and $S_j = \emptyset$. That is, if process P_i has color j at s, then $pc_i \in \{rem, first\text{-}door, reserve, test, leave\text{-}exit\}$. Every process satisfying $pc = leave\text{-}exit$ eventually executes rem_i and sets $pc := rem$. If a process satisfying $pc = rem$ receives a request for the resources, it executes try_i and sets $pc := first\text{-}door$. Every process satisfying $pc = first\text{-}door$ eventually executes $first\text{-}door_i$. This causes either a transition that the process increases $count(j)$ by 1 and becomes an element of S_j, or a transition that the process sets $pc := reserve$. Every process satisfying $pc = reserve$ eventually executes $reserve_i$ and sets $pc := test$. Every process satisfying $pc = test$ eventually executes $test_i$, increases $count(j)$ by 1, and becomes an element of S_j. Consequently, after system state s, if a process that has color j receives a request for the resources, then it increases $count(j)$ by 1 exactly once by executing either $first\text{-}door_i$ or $test_i$. Note that processes that have color $j' \neq j$ cannot increase $count(j)$.

It follows from these facts that after system state s, $count(j)$ is kept to be equal to $|S_j|$ until some process executes $inverse_i$. Observe that after system state s, exactly one process in S_j whose num is equal to 0 can execute $elect_i$. Since that process inserts j into queue $COLORQ$ during the execution of $elect_i$, it must be P_{GT}. It is not difficult to see that P_{GT} executes in turn $elect_i, second\text{-}door_i, keep_i$, and $inverse_i$. By the execution of $inverse_i$, the value of $count(j)$ becomes negative but its absolute value remains the same. Observe that from the conditions of $first\text{-}door_i$ and $test_i$, no process can become an element of S_j after the execution of $inverse_i$ until $count(j)$ becomes nonnegative.

At some later point after the execution of $inverse_i$, every process in S_j executes in turn $crit_i, exit_i$, and $disposal_i$. If $count(j) < -1$ holds true at the execution of $disposal_i$, then the executing process increases $count(j)$ by 1 and proceeds to rem_i. This results in that the process is no longer an element of S_j. In this way, $count(j)$ is increased and the size of S_j decreases. Therefore, when $|S_j|$ is equal to 1, the value of $count(j)$ must be -1. After that point, the last process in S_j, i.e., P_{GE} executes in turn $disposal_i, remove\text{-}queue_i, reset_i$, and rem_i. After the execution of $disposal_i$, $count(j) = 0$ holds true. After the execution of $reset_i$, $S_j = \emptyset$ holds true. □

Lemma 2 *During the execution of the color-grouping algorithm, if processes that have color j are in C, then j is first on $COLORQ$.*

Theorem 1 *For any execution of the color-grouping algorithm, there is no reachable system state in which processes that have different colors are in C.*

Lemma 3 *Let T_g be the time from when some color is first on $COLORQ$ until the color is deleted from $COLORQ$. For any execution of the color-grouping algorithm, T_g is $O((m + k)l + c)$.*

Lemma 4 *Let j be an arbitrary color. Let T_c be the time from when the value of shared variable $count(j)$ is negative until it has nonnegative value. For any execution of the color-grouping algorithm, T_c is at most $(3m + k + 1)l + c$.*

Lemma 5 *Let T_1 be the time from when any particular process enters T until pc of the process is equal to elect. For any execution of the color-grouping algorithm, T_1 is at most $(5m + k + 1)l + c$.*

Lemma 6 *Let T_2 be the time from when pc of any particular process is equal to elect until it enters C. For any execution of the color-grouping algorithm, T_2 is $O(k(m + k)l + kc)$.*

Theorem 2 *For any execution of the color-grouping algorithm, the time from when any particular process enters T until it enters C is $O(k(m+k)l+kc)$.*

4 Concluding Remarks

We have proposed a distributed algorithm for the resource-allocation problem on the asynchronous read-modify-write shared memory model. It is clear that the algorithm guarantees the well-formedness. From Theorem 1, the algorithm satisfies the exclusion condition. From Theorem 2, the progress for the trying region, progress for the exit region, lockout-freedom for the trying region, and lockout-freedom for the exit region are guaranteed by the algorithm. Hence, the proposed algorithm solves the resource-allocation problem.

For any execution of the proposed algorithm, the time from when any particular process enters its trying region until it enters its critical region is $O(k(m+k)l+kc)$. Compared with the maximum waiting time by the algorithm in [5], we can conclude that the proposed algorithm achieves an improvement on waiting time.

For further investigation, we pose the following problems.

- Design a distributed algorithm for the resource-allocation problem on the asynchronous single-writer/multi-reader shared memory model.
- Construct a fault-tolerant algorithm for the resource-allocation problem on the asynchronous read-modify-write shared memory model.
- Devise a distributed algorithm for the drinking philosophers problem on the asynchronous read-modify-write shared memory model.

References

[1] K. M. Chandy and J. Misra, "The drinking philosophers problem", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 4, pp. 632–646, 1984.

[2] M. Choy and A. K. Singh, "Efficient fault tolerant algorithms for resource allocation in distributed systems", In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pp. 593–602, 1992.

[3] V. K. Garg, "Elements of Distributed Computing", John Wiley & Sons, 2002.

[4] N. A. Lynch, "Upper bounds for static resource allocation in a distributed system", *Journal of Computer and System Sciences*, Vol. 23, No. 2, pp. 254–278, 1981.

[5] N. A. Lynch, "Distributed Algorithms", Morgan Kaufmann, 1996.