非同期共有メモリシステムにおける適応型繰り返し改名アルゴリズム

梅谷 真也 † 井上 美智子 † 増澤 利光 ‡ 藤原 秀雄 †

† 奈良先端科学技術大学院大学 情報科学研究科

‡ 大阪大学大学院 基礎工学研究科

あらまし 本論文では，非同期共有メモリシステム上で，効率良く名前の獲得と解放を繰り返し行う分散アルゴリズム (改名アルゴリズム) を提案する．N 個のプロセスは $1, 2, \cdots, N$ の範囲の一意な名前を最初に持っているが，M-改名アルゴリズムを実行することで，新しく $1, 2, \cdots, M$ の範囲の一意な名前を獲得する．

従来の最良の $O(k^2)$-改名アルゴリズムとして，Afek らがステップ計算量 $O(k^3)$，空間計算量 $O(n^3N)$ のアルゴリズムを提案した [1]．ここで，k はポイント競合度と呼ばれ，名前を獲得する際に，同時にステップを実行したり名前を保持しているプロセスの数の最大値を示す．また，n は k の上界である．さらに，変数の値が非有界となるが，ステップ計算量 $O(k^2 \log k)$，空間計算量 $O(n^3N)$ のアルゴリズムも提案されている [1]．本論文ではこれら従来の改名アルゴリズムより効率の良い，ステップ計算量 $O(k^2)$，空間計算量 $O(n^2N)$ の $O(k^2)$-改名アルゴリズムを提案する．

Adaptive Long-lived Renaming Algorithm in the asynchronous shared memory

Shinya Umetani† Michiko Inoue† Toshimitsu Masuzawa‡ Hideo Fujiwara†

† Graduate School of Information Science, Nara Institute of Science and Technology

‡ Graduate School of Engineering Science, Osaka University

Abstract. This paper presents an adaptive long-lived renaming algorithm in the asynchronous shared memory system. The system consists of N asynchronous process, and each process initially has a distinct name in the range $\{1, 2, \cdots, N\}$. A M-renaming algorithm assigns new unique name in the range $\{1, 2, \cdots, M\}$ to any process.

The previous best $O(k^2)$-renaming algorithm is the algorithm with $O(k^3)$ step complexity and $O(n^3N)$ space complexity presented by Afek et. al [1], where k is the point contention and n is upper bound of k. The point contention is the maximum number of processes that actually take steps or hold a name while the new name is being acquired. They also presented the algorithm with $O(k^2 \log k)$ step complexity and $O(n^3N)$ space complexity under the condition where unbounded values are allowed. The step complexity of our algorithm is $O(k^2)$, and space complexity is $O(n^2N)$. That is, our $O(k^2)$-renaming algorithm is more efficient than two previous algorithms.

1 Introduction

An *asynchronous read/write shared memory model* consists of asynchronous processes and shared registers. Each process has a distinct identifier, and communicates via read and write operatins on shared registers. We consider a *long-lived M-renaming problem* in this model. In the problem, every process repeatedly acquires a new name in the range $\{1, 2, \cdots, M\}$, and releases it after the use. The problem requires that no two process keep the same name concurrently, and renaming algorithm is required to have small name space and low complexity.

Recently, the algorithms where the step complexities depend on only the *contention*, the number of the active processes which actually participate in the algorithm, were proposed. Such algorithms are called to be *adaptive*. In adaptive algorithms, the number of actually active processes is unknown in advance. The adaptive renaming algorithm is very useful if the number of active process is much smaller than the number of total process which have a potential for participation. Since the complexities of most distributed algorithms depend on the name space of processes, we can reduce the complexities by using the renaming algorithm to reduce the name space.

Table 1 shows the results on adaptive long-lived renaming algorithms. Afek et al. [2] proposed adaptive long-lived renaming algorithm, which is adaptive to the *point contention* and use unbounded memory, where the point contention, denoted k, is the maximum number of processes being concurrently active at some point in the execution. Afek et al. next proposed following three long-lived renaming algorithms [1] which adapt to the point contention. Their algorithms are a $(2k^2 - k)$-renaming with $O(k^2 \log k)$ step complexity and $O(n^3 N)$ space complexity using unbounded values, a $(2k^2 - k)$-renaming with $O(k^3)$ step complexity and $O(n^3 N)$ space complexity using bounded values, and $(2k-1)$-renaming with $Exp(k)$ step complexity and $O(n^3 N)$ space complexity using unbounded values. Attiya et al. [3] proposed a long-lived $(2k-1)$-renaming algorithm which adapts to point contention with $O(k^4)$ step complexity.

In this paper, we present a long-lived $(2k^2 - k)$-renaming algorithm that adapts to the point contention with $O(k^2)$ step complexity and $O(n^2 N)$ space complexity using bounded values. That is, our algorithm is more efficient than above $(2k^2 - k)$-renaming algorithms.

2 Preliminaries

Our computation model is an asynchronous read/write shared memory model [4]. A shared memory model consists of N processes, $p_0, \cdots, p_{N-1}$ and a set of registers shared by the processes. The processes communicate each other by reading from and writing to shared registers. We assume *multi-writer-multi-reader* registers, that is, each process can read from and write to any register.

In the *long-lived M-renaming* problem, processes repeatedly acquire and release distinct names in the range $\{1, 2, \cdots, M\}$. A renaming algorithm provides two procedures $\mathsf{getName_i}$ and $\mathsf{releaseName_i}$ for each process p_i. A process p_i uses $\mathsf{getName_i}$ to get a new name, and uses $\mathsf{releaseName_i}$ to release it. Each process alternates between invoking $\mathsf{getName_i}$ and $\mathsf{releaseName_i}$, starting with $\mathsf{getName_i}$.

An execution of an algorithm is a (possibly infinite) sequence of register operations and invocations and returns of procedures where each process follows the algorithm. Let α be some execution of a long-lived renaming algorithm, and let α' be some finite prefix of α. Process p_i is *active* at the end of α', if α' includes an invocation of $\mathsf{getName_i}$ without a return from the matching $\mathsf{releaseName_i}$. Process p_i which is active at the end of α' can either be *trying* to get a new name, that is, p_i has not yet returned from $\mathsf{getName_i}$, or *holding* a name y, that is, p_i has already returned from $\mathsf{getName_i}$. In the latter case, we say that p_i *holds a name y* at the end of α' if the last invocation of $\mathsf{getName_i}$ returned y. A long-lived renaming algorithm should guarantee the following *uniqueness* : If active processes p_i and p_j $(j \neq i)$ hold names y_i and y_j, respectively, at the end of α', then $y_i \neq y_j$.

The *contention* at the end of α', denoted

Step complexity	Name space	Space complexity	Value size	Reference
$O(k^2 \log k)$	$2k^2 - k$	unbounded	bounded	[2]
$O(k^2 \log k)$	$2k^2 - k$	$O(n^3 N)$	unbounded	[1]
$O(k^3)$	$2k^2 - k$	$O(n^3 N)$	bounded	[1]
$Exp(k)$	$2k - 1$	$O(n^3 N)$	unbounded	[1]
$O(k^4)$	$2k - 1$	unbounded	unbounded	[3]
$O(k^2)$	$2k^2 - k$	$O(n^2 N)$	bounded	this paper

Table 1: Adaptive renaming algorithms.

$Cont(\alpha')$, is the number of active processes at the end of α'. Let β be a finite interval of α, that is, $\alpha = \alpha_1 \beta \alpha_2$ for some α_1 and α_2. The *point contention* of β, denoted $PntCont(\beta)$, is the maximum contention over all prefixes $\alpha_1 \beta'$ of $\alpha_1 \beta$.

The name space which is obtained by using renaming algorithm is adaptive to the point contention if there is a function F, such that the name obtained in an interval β of $\mathsf{getName_i}$, is in the range $\{1, 2, \cdots, F(PntCont(\beta))\}$. The step complexity of a renaming algorithm is adaptive to point contention if there is a bounded function S, such that the number of steps performed by p_i in any interval β of $\mathsf{getName_i}$ and in the matching $\mathsf{releaseName_i}$ is at most $S(PntCont(\beta))$.

3 Renaming Algorithm

Our renaming algorithm is based on the $(2k^2 - k)$-renaming algorithm presented in [1], which is a long-lived renaming algorithm and adapts to point contention k with $O(k^3)$ step complexity using bounded memory and bounded values. The major difference between this algorithm and our algorithm is detail of procedures interleaved_sc_sieve, leave and clear called in the top level procedures, getName and releaseName, which is shown in Algorithm 1.

Our renaming algorithm uses a sequence of *sieves*, numbered $1, 2, \cdots, 2n$, and each sieve has $2N$ *copies*, numbered $0, 1, \cdots, 2N-1$, where one copy is work space for processes which visit the sieve concurrently. The first component of the variable $sieve[s].count$ is changed to $0, 1, \cdots, 2N-1, 0, 1, \cdots$, cyclically, and a shared variable $sieve[s].count$ designates the current copy of the sieve s. We can associate a *round* with the value of $sieve[s].count$ which means how many times the variables is updated to the current value. If a process sees $sieve[s].count$ with a round r, we say the process uses the designated copy in the round r.

In the procedure $\mathsf{getName_i}$, a process p_i visits a sequence of sieves one after the other until it *wins* in some sieve. If a process p_i visiting a sieve s satisfies some conditions (Line 9), it *enters* one copy and obtains a set W of process identifiers. If W is a non-empty set including its identifier, p_i wins in the sieve s, and p_i gets a new name $\langle s$, the rank of p_i in $W\rangle$ (Line 13).

In the procedure $\mathsf{releaseName_i}$, a process p_i leaves the copy which p_i got a name to show that p_i released the name.

If p_i notices that all candidates leave this copy, p_i initializes the copy to reuse it in the next round by invoking the procedure clear (Line 15 and 20). A Boolean variable $sieve[s].allDone[c]$ is used as a signal that a round in the copy c of the sieve s has been finished. The value $nextDB$ differs with the parity of the round (Line 7 and 8). However, some slow processes excluding W may still work in the copy after the initialization started. Therefore, after every operation to shared registers, each process checks whether the copy has been finished or not. If the process notices that the copy has been finished, it initializes the last modified register and leaves the sieve. This mechanism is implemented by *interleave*.

In the procedure sc_sieve, a process enters a copy c of a sieve s, if all names assigned from the previous copy $c - 1 \bmod 2N$ are released by checking a variable $sieve[s].allDone[c-1 \bmod 2N]$, and the current copy c is free by checking a variable $sieve[s].inside[c]$ (Line 30). The Boolean variable $sieve[s].allDone[c]$ is changed after all names assigned from c

Algorithm 1: Procedure of renaming algorithm : part I.

```
Shared variables :
    sieve[1,...,2n−1] {
      count : ⟨integer,Boolean⟩, initially ⟨0,0⟩;
      status[0,...,N−1] : Boolean, initially false;
      inside[0,...,2N−1] : Boolean, initially false;
      allDone[0,...,2N−1] : Boolean, initially false;
      list[0,...,2N−1] {
        mark[0,...,n−1] : Boolean, initially false;
        view[0,...,n−1] : set of ⟨id,integer⟩, initially ⊥;
        id[0,...,n−1] : id, initially ⊥;
        X[0,...,n−1] : integer, initially ⊥;
        Y[0,...,n−1] : Boolean, initially false;
        done[0,...,n−1] : Boolean, initially false;
    }}

Non-shared Global variables :
    nextC,c : integer, initially 0;
    nextDB,dirtyB : Boolean, initially false;
    W : set of ⟨id,integer⟩, initially ∅;
    s : integer, initially 0;
    sp : integer, initially ⊥;

procedure getName()
1   s = 0;
2   while (true) do
3     s++;
4     sieve[s].status[i] = active;
5     ⟨c, dirtyB⟩ = sieve[s].count;
6     nextC = c + 1 mod 2N;
7     if (nextC = 0) then nextDB = not dirtyB;
8     else nextDB = dirtyB;
9     if ((nextC mod N = i) or
          (sieve[s].status[nextC mod N] = idle)) then
10      W = interleaved_sc_sieve(sieve[s], nextC, nextDB);
11      if (⟨p_i, sp⟩ ∈ W for some sp) then
12        sieve[s].count = ⟨nextC, nextDB⟩;
13        return ⟨s,rank of p_i in W⟩;
14      else-if (sieve[s].allDone[nextC] = nextDB) then
15        clear(sieve[s], nextC);
16    sieve[s].status[i] = idle;
17  od;

procedure releaseName()
18  leave(sieve[s], nextC, nextDB);
19  if (sieve[s].allDone[nextC] = nextDB) then
20    clear(sieve[s], nextC);
21  sieve[s].status[i] = idle;
```

of s are released. The Boolean variable $sieve[s].inside[c]$ is changed after some processes enter c of s.

If the process can enter the copy c, it tries to register in c by invoking the procedure register (Line 32). If it can register, it scans c to obtain a snapshot of processes which has registered in c by invoking the procedure partial_scan (Line 34). This can be achieved by invoking the procedure collect twice which returns a set of process identifiers. We call such a set *view*. If two views are identical, it returns the view as a snapshot, otherwise it returns an empty set.

Then, the process find the minimum snapshot W of processes by invoking the procedure candidates (Line 35). If a process obtains a non-empty snapshot W, W is a set of candidates of winners in s.

To implement the procedures register and collect, we use the *collect list*. The collect list consists of $2n$ *splitters*, and each splitter has a distinct level in the range $\{0, 1, \cdots, 2n-1\}$ [5]. The splitter returns either **stop**, **next** or **abort**. Algorithm 4 shows procedures register, collect and splitter.

The procedures register and collect are adaptive to *total contention*, where the total contention is the number of active processes in an execution of these procedures. These procedures are executed by only the processes which entered the same copy concurrently. This means they are concurrently active in some point, and in a round, the processes which executes the procedures register and collect in the copy c of the sieve s are only them. That is, we can use these procedures as procedures which are adaptive to point contention.

Algorithm 2: Procedures renaming algorithm: part II.

Non-shared Global variables :
    last_modified : points to last shared variable modified by p_i;
        // *last_modified* is assumed to be updated immediately before the write.
    mysplitter : integer, initially $\perp$;

procedure interleaved_sc_sieve($sieve, nextC, nextDB$)
    // interleave is a two part construct. Part I of the interleave is executed after every read or write
    // to a shared variable in Part II, the sc_sieve() and any procedure recursively called from sc_sieve().
$last\_modified = \perp$;
interleave { // Part I
  if ($sieve.allDone[nextC] = nextDB$) then
    if ($last\_modified \neq \perp$) then write initial value to *last_modified*;
    return $\emptyset$;   // abort current sc_sieve(), s, and continue to next sieve.
}{ // Part II
  return sc_sieve($sieve, nextC, nextDB$);
}

procedure sc_sieve($sieve, nextC, nextDB$)
if (previousFinish($sieve, nextC, nextDB$) and $sieve.inside[nextC]$ = **false**) then
  $sieve.inside[nextC]$ = **true**;
  $mysplitter$ = register($sieve.list[nextC]$);
  if ($mysplitter \neq \perp$) then
    $sieve.list[nextC].view[mysplitter]$ = partial_scan($sieve.list[nextC]$);
    W = candidates($sieve, nextC$);
    if ($\langle p_i, mysplitter\rangle \in W$) then return W;
    $sieve.list[nextC].done[mysplitter]$ = **true**;
    W = candidates($sieve, nextC$);
    leave($sieve, nextC, nextDB$);
return $\emptyset$;

procedure previousFinish($sieve, nextC, nextDB$)
if ($nextC \neq 0$ and $sieve.allDone[nextC - 1 \bmod 2N] = nextDB$) then return **true**;
if ($nextC = 0$ and $sieve.allDone[2N-1] \neq nextDB$) then return **true**;
return **false**;

4 Correctness

4.1 Correctness of Collect List

A implementation of a splitter uses two shared variables, X and Y. Initially, $X = \perp$ and Y = **false**. A process executing the procedure splitter first writes its identifier into X and then reads Y. If Y = **true**, the process returns **abort**. Otherwise, the process writes **true** into Y and checks X. If X still contains its identifier, the process returns **stop**. Otherwise, the process returns **next**. By the algorithm, we have the following property of the splitter.

Lemma 1 *If s processes take access to the same splitter concurrently, the following conditions hold: (1) at most one process obtains* **stop***, (2) at most $s-1$ processes obtain* **abort***, and (3) at most $s-1$ processes obtain* **next**.

To prove the correctness and complexity of the collect list, let k' be the total contention, where the number of processes which enter a copy of a sieve in a round and invoke register and collect.

Lemma 2 *If the level of a splitter v in the collect list is l, $0 \leq l \leq k'$, then at most $k'-l$ processes take access to v.*

Proof : We prove this lemma by induction on l, the level of v. In the base case, $l = 0$, the lemma trivially holds since at most k' processes are active in the collect list. For the induction step, suppose that the lemma holds for a splitter u with level l, $0 \leq l < k'$, and consider some splitter v with level $l+1$. The level of u is l, and by the inductive hypothesis, at most $k'-l$ processes take access to u. Then, the property (3) of the splitter (lemma 1) implies that at most $k'-l-1$ of the processes

Algorithm 3: Procedures renaming algorithm: part III.

procedure partial_scan($list$)
V_1 = collect($list$);
V_2 = collect($list$);
if ($V_1 = V_2$) then return V_1;
else return $\emptyset$;

procedure candidates($sieve, copy$)
$sp = 0$; $V = \emptyset$;
while ($sieve.list[copy].mark[sp] = \textbf{true}$) do
  if ($sieve.list[copy].view[sp] \neq \perp$) then $V = V \cup \{sieve.list[copy].view[sp]\}$;
  sp++;
od;
if $V = \emptyset$ then return $\emptyset$;
$U = \min\{view | view \in V \text{ and } view \neq \emptyset\}$;
if $U \neq \emptyset$ and for every $\langle p_j, sp\rangle \in U$, $sieve.list[copy].view[sp] \supseteq U$
    or $sieve.list[copy].view[sp] = \emptyset$ then return U;
else return $\emptyset$;

procedure clear($sieve, nextC$)
$sieve.inside[nextC] = \textbf{false}$; $sp = 0$;
while ($sieve.list[nextC].mark[sp] = \textbf{true}$) do
  write initial value to a splitter sp in $sieve.list[nextC]$;
  sp++;
od;

procedure leave($sieve, nextC, nextDB$)
$sieve.list[nextC].done[mysplitter] = \textbf{true}$;
if $W \neq \emptyset$ and for every $\langle p_j, sp\rangle \in W$, $sieve.list[nextC].done[sp] = \textbf{true}$ then
  $sieve.allDone[nextC] = nextDB$;

obtain **next** at u and take access to v. ■

By Lemma 2 and the algorithm, when a process executes register, it stops or aborts in a splitter with level less than or equal to $k' - 1$. By the property (1) of the splitter (lemma 1), at most one process stops in each splitter. Therefore, we have the following lemma.

Lemma 3 *Each process which obtains* **stop** *by invoking* splitter *writes its identifier in the splitter with level* $\leq k' - 1$*, and no other process writes its identifier in the same splitter.*

By Lemma 3, procedure register and collect visits at most k' splitters, each splitter requires a constant number of operations.

Theorem 4 *The step complexities of the procedure* register *and* collect *are* $O(k')$.

By the algorithm, a process p_i once register in a splitter sp_i by invoking the procedure register, processes never update the variable $id[sp_i]$. And the procedure collect scans a collect list sequentially. By the above properties of collect list, we have the following lemma.

Lemma 5 *Assume a* collect *operation* cop_1 *executed by* p_i *returns* V_1*, and a* collect *operation* cop_2 *executed by* p_j *returns* V_2*. If* cop_2 *starts after* cop_1 *finishes, then* $V_1 \subseteq V_2$.

The collect returns a view consisting of all process identifiers which registered before invoking collect and some process identifiers which register concurrently with the execution of collect. Let V be a set obtained by an execution of partial_scan, and let V_1 and V_2 be non-empty views obtained by consecutive two invocations of collect in the partial_scan. If a process obtains the identical views, that is $V_1 = V_2$ the set V is a snapshot of processes which have registered at some point between two collects.

Lemma 6 *For every non-empty sets* V_1 *and* V_2 *which is obtained by invoking the procedure* partial_scan*, either* $V_1 \subseteq V_2$ *or* $V_2 \subseteq V_1$.

Algorithm 4: Procedures of collect list.

```
procedure register(list)
  sp = 0;
  while (true)
    list.mark[sp] = true;
    move = splitter(list, sp);
    if (move = next) then
      sp++;
    if (move = abort) then
      return ⊥;
    if (move = stop) then
      list.id[sp] = p_i;
      return sp;
  od;

procedure collect(list)
  sp = 0; V = ∅;
  while (list.mark[sp] = true)
    if (list.id[sp] ≠ ⊥) then V = V ∪ {⟨list.id[sp], sp⟩}
    sp++;
  od;
  return V;

procedure splitter(list, currentsp)
  list.X[currentsp] = p_i;
  if (list.Y[currentsp] = true) then return abort;
  list.Y[currentsp] = true;
  if (list.X[currentsp] = p_i) then return stop;
  else return next;
```

4.2 Correctness of Renaming Algorithm

We briefly show the correctness of our renaming algorithm.

By the same access control as the renaming algorithm presented in [1], our algorithm guarantees that all processes entering a copy in some round leave and the copy is initialized before the copy is used in the next round. The behavior of some copy in some round is independent of the behavior of the previous rounds in the copy. Therefore, the following lemmas concern a copy in one round and it is enough to show the procedure partial_scan and candidates work well on behalf of latticeAgreement and candidates in [1].

Lemma 7 *If W_1 and W_2 are non-empty views returned by invocations of* candidates(s, c) *for the same copy c of the same sieve s in the same round then $W_1 = W_2$.*

Proof : We prove this lemma by contradiction. Assume $W_1 \neq W_2$. By lemma 6, $W_1 \subset W_2$ or $W_2 \subset W_1$. We assume $W_1 \subset W_2$ without less of generality. A snapshot returned by candidates is a snapshot obtained by using partial_scan in the same copy in the same round. Let p_i be a process which obtains W_1 by partial_scan, and p_j be a process which obtains W_2 by candidates. Since p_i is the only process which updates the variable $s.list[c].view[sp]$ in this round, the value of $s.list[c].view[sp]$ must be the initial value $\perp$ or W_1. However, p_j sees that $s.list[c].view[sp] \supseteq W_2$ or $s.list[c].view[sp] = \emptyset$. A contradiction. ■

If an invocation of candidates(s, c) by process p_i returns a non-empty view containing itself, p_i is a *winner* in copy c of sieve s. Lemma 7 implies following lemma.

Lemma 8 *If process p_w is a winner in copy c of sieve s in some round, then p_w appears in every non-empty view returned by an invocation of* candidates(s, c) *in this round.*

Process p_i is *inside* copy c of sieve s after it executes line 31 with c. A process inside copy c of sieve s is *done* after it assigns **true** to $sieve.list[c].done[sp]$ (in line 37, if it is a winner, or in line 62, otherwise).

Lemma 9 *If process p_i is inside copy c of sieve s in some round, all winners of the previous copy $c - 1 \bmod 2N$ of sieve s are done in this round.*

By Lemma 8 and 9, we can show the following uniqueness.

Lemma 10 *If active processes p_i and $p_j (j \neq i)$ hold names y_i and y_j, respectively, at the end of some finite prefix of some execution, then $y_i \neq y_j$.*

The following two lemmas are used to give an upper bound of the number of sieves to which each process visits.

Lemma 11 *If one or more processes enter a copy c of a sieve s in some round, at least one process obtains a snapshot by invoking* partial_scan *in c in this round.*

Proof : We prove the lemma by contradiction. Assume that no process obtains a snapshot. In this case, no process writes non-empty set to a variable $s.list[c].view[sp]$, obtains non-empty set by candidates, and writes $nextDB$ to the variable $s.allDone[nextC]$. Since a copy is initialized after $s.allDone[nextC]$ is set to $nextDB$, no process initializes the copy. Let p_i be the last process which writes its identifier to a variable $s.list[c].id[sp]$ of some splitter sp in the copy c in the procedure register. The process p_i then executes collect twice in partial_scan. Since a set of processes which have registered does not changes after p_i registered, p_i can obtains a snapshot. A contradiction. ■

Lemma 12 *If one or more processes enter a copy c of a sieve s in some round, at least one process wins in c in this round.*

Proof : Lemma 11 shows that at least one process obtains a snapshot by invoking the procedure partial_scan in the copy c in this round. Let W be the minimum snapshot obtained in c in this round, and p_i be the last process in W which writes a value to $s.list[c].view[sp]$ in some splitter sp. Since W is the minimum snapshot, every process p_j in W obtains a snapshot W' not smaller than W or fails to obtain a snapshot. Therefore p_j writes a view W' in its splitter such that $W \subseteq W'$ or $W = \emptyset$. The process p_i can see these values in candidates and return W including p_i. That is, p_i wins in c in this round. ■

We use Lemma 12 to show the following lemma by the similar way to Lemma 3.5 in [1].

Lemma 13 *Every process p wins in sieve at most $2k-1$, where k is the point contention of p's interval of* getName.

The step complexity of our algorithm is as follows. In getName, each process p_i visits to at most $2k-1$ sieves, and takes access to and enters at most one copy in each sieve. For each copy, p_i invokes one register, two collect, and at most one clear. Each procedure has $O(k)$ step complexity, and therefore, total step complexity is $O(k^2)$. The algorithm uses $2n-1$ sieves, 2N copies of each sieve, and $O(n)$ registers for each copy. Therefore, the space complexity is $O(n^2N)$.

Theorem 14 *Our algorithm solves the point contention adaptive long-lived $(2k^2-k)$-renaming problem with $O(k^2)$ step complexity and $O(n^2N)$ space complexity using bounded values.*

5 Conclusion

We have presented a long-lived $(2k^2-k)$-renaming algorithm that adapts to point contention k and uses bounded values. The step complexity is $O(k^2)$ and the space complexity is $O(n^2N)$ where n and N are upper bound of k and ther number of processes, respectively.

Our future work is to improve our algorithm. We would like to develop an efficient long-lived $(2k-1)$-renaming algorithm which is adaptive to point contention with polynomial step complexity and uses bounded memory.

References

[1] Y.Afek, H.Attiya, A.Fouren, G.Stupp, and D.Touitou. Adaptive long-lived renaming using bounded memory. 1999. Available at www.cs.technion.ac.il/~hagit/pubs/AAFST99disc.ps.gz.

[2] Y.Afek, H.Attiya, A.Fouren, G.Stupp, and D.Touitou. Long-lived renaming made adaptive. *In Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 91–103, 1999.

[3] H.Attiya and A.Fouren. Polynomial and adaptive long-lived (2k-1)-renaming. *In Proc. 14th Int. Symp. on Dist. Comp.*, 2000.

[4] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[5] H.Attiya and A.Fouren. An adaptive collect algorithm with applications. 1999. Available at www.cs.tehnion.ac.il/~hagit/pubs/AF99ful.ps.gz.