行政院國家科學委員會補助專題研究計畫成果報告

※※※※※※※※※※※※※※※※※※※※※※※※※
※ ※
※ **互斥問題之熱點衝突與路徑長度極小化** ※
※ ※
※※※※※※※※※※※※※※※※※※※※※※※※※

計畫類別：¦**個別型計畫** □**整合型計畫**

計畫編號：NSC 89－2213－E－009－027－

執行期間：88 **年** 8 **月** 1 **日至** 89 **年** 7 **月** 31 **日**

計畫主持人：黃 廷 祿

本成果報告包括以下應繳交之附件：

□**赴國外出差或研習心得報告一份**

□**赴大陸地區出差或研習心得報告一份**

□**出席國際學術會議心得報告及發表之論文各一份**

□**國際合作研究計畫國外研究報告書一份**

執行單位：國立交通大學資訊工程學系

中 華 民 國 89 **年** 10 **月** 26 **日**

行政院國家科學委員會專題研究計畫成果報告
互斥問題之熱點衝突與路徑長度極小化
Minimizing Hot-spot contention and critical path length for the mutual exclusion problem

計畫編號：NSC 89-2213-E-009-027
執行期限：88 年 8 月 1 日至 89 年 7 月 31 日
主持人：黃廷祿　國立交通大學資訊工程學系
E-mail address: tlhuang@csie.nctu.edu.tw

一、中文摘要

互斥問題為古典分散式演算法常被探討之問題。針對熱點衝突之度量與路徑長度兩者之積求其最佳化之演算法早已存在，但這些演算法僅允許固定的二元樹狀結構，因而無從調節兩個度量之互相消長。吾人提出較具調節彈性之演算法，而不失其最佳化之特性。

關鍵詞：互斥問題、熱點衝突、分散式演算法

Abstract

Optimal solutions exist for the mutual exclusion problem minimizing the product of two measures: 1) hot spot contention, or maximal number of pending operations for any individual variable in any execution of the algorithm, and 2) critical path length, or maximal number of remote memory references executed by a process in one life cycle. That the two measures constitute a trade-off for the problem was also known. However, the existing solutions achieve optimality by enforcing a binary tournament tree for the contending processes and therefore permit no flexibility in the trade-off between the two measures. We present the first algorithm that permits flexibility in fixing a desirable value of hot spot contention while keeping critical path length at the lowest possible level.

Keywords: Mutual Exclusion, Hot Spot Contention, Distributed algorithms

二、緣由與目的

Memory contention, the extent to which processes access the same location simultaneously, has long been recognized by practitioners as one of the key factors that slow down the execution of a shared memory program, especially when the number of processors is large. *Hot spot contention*, the maximal number of pending operations for any individual variable in any execution of the algorithm, is of particular interest since it is a simple measure of the worst contention at any instant in the execution. To alleviate hot spot contention, techniques of tournament trees to limit the number of contending requests had been widely used, at the expense of increasing *critical path length*, which is the maximal number of memory operations executed by any process in one life cycle. Despite its importance in practice, until the recent formal model proposed by Dwork et al. [1], complexity analysis of memory contention for asynchronous shared

memory algorithms had not been adequately addressed. With their formal model, a tight bound $\Theta(\log n)$, where n is the number of processes, on the *product* of critical path length and hot spot contention, for the *one-shot* mutual exclusion problem was proved. The product can be considered as an estimate of the worst-case protocol delay for any process in one life cycle. The problem, however, is merely a part of the mutual exclusion problem. We consider in this article the mutual exclusion problem, the solution to which is a necessity for resource sharing in any multiprocessor system. The previous lower bound $\Omega(\log n)$ for the sub-problem is necessarily a lower bound for the full problem. Since there exist already binary tournament tree-based mutual exclusion algorithms that requires $O(\log n)$ on the product, our algorithm is merely a new member, not the first, of the class that reaches optimality.

It is premature to suggest that one measure clearly dominates the other in affecting system performance in practice, algorithms that permit flexibility in choosing an appropriate value for hot spot contention while keeping critical path length at the lowest possible level are of general interest. Prior optimal solutions [4, 5] require that the tournament tree be binary, resulting in the longest critical path length in the optimal trade-off. Our algorithm not only permits flexibility in fixing a value for hot spot contention but also retains optimality in minimizing the product of the two measures.

The algorithm assumes that both *compare&swap* and *fetch&store* primitives are available. Prior optimal solutions assume only atomic read/write registers. Whether the lack of flexibility in their part is inevitable without the support of read-modify-write registers is interesting but not yet investigated.

Dwork et al. [1] proposed a formal model of memory contention for asynchronous shared memory algorithms. A special type of algorithms is defined as a wait-free protocol if it requires no waiting for other processes in all executions. Three measures intended to facilitate efficiency analysis of any such protocol are *contention*, *hot spot contention* and *critical path length*. Contention is the ratio of the worst-case execution's contention cost divided by n. An execution's contention cost are the sum over all processes of the numbers of memory access stalls that were incurred by each process in its execution of the protocol. Hot spot contention is the maximal number of pending operations for any variable in any execution. It is meant to measure the contribution of an individual variable to the execution's contention cost. Critical path length is the maximal number of memory operations executed by any process in one life cycle. The product of hot spot contention and critical path length is intuitively the worst-case protocol delay incurred by any process in one life cycle.

For wait-free protocols using read-modify-write primitives, the above mentioned memory contention model is accurate enough to make useful comparisons, yielding several lower bound and tight bound results for some well-known problems.

However, there are important problems that admit no wait-free solutions even when read-modify-write primitives are available. Mutual exclusion and barrier synchronization, among others, are problems whose solutions must use some type of busy waiting, if shared memory is the sole medium for synchronization.

For those synchronization problems that entail busy waiting, techniques of looping only on local variables, or **local spin**, have proved effective in shared memory systems that allow each processor to declare part of the global address space as its local area which is mapped to some fast local memory locations for the processor. If each processor spins only on its local variables, there would be no contention cost incurred by the spinning since the local areas are non-overlapping between any pair of processors. One remote write is sufficient to wake up a waiting processor. In order to facilitate efficiency analysis for these algorithms in real systems, we need to review how the measure of the product (the protocol delay) is taken. The delay experienced by a process in a busy waiting loop depends largely on the length of the application that other processes are engaging at the moment. Little can be done for protocol designers to reduce the length of waiting delay. Therefore we should not count the number of all memory references (local or remote) of the worst-case execution as the critical path length (which can be arbitrarily large). We should count only the number of references to remote variables (which can be local variables of others) as the critical path length.

```
compare&swap(r: public register, old, new :
                value) returns(value)
    previous := r
    if previous = old
        then r:= new
    fi
    return previous
```

Figure 1: Compare&swap promitive.

```
type q-node = record
    wait : Boolean     int   True;
    direct : Boolean   int   False;
    hold : Boolean     int   True;
```

Figure 2: Per Process data structures for the base algorithm.

三、結果與討論

It suffices to note that the lower bound for one-shot mutual exclusion is necessarily the lower bound for the full mutual exclusion problem because a solution for the latter must also solve the former. Proof of the lower bound result for one-shot mutual exclusion can be found in Dwork et al. [1].

Figure 3 is the base algorithm that will be reused in the full-fledged algorithm. The base algorithm decented from the circular list-based algorithm (CL algorithm) by Fu and Tzeng [3], with two important improvements: deadlock freedom in entering C region and lockout freedom in entering R region.

Figure 2 shows the data structure of the memory space that is allocated for each process.

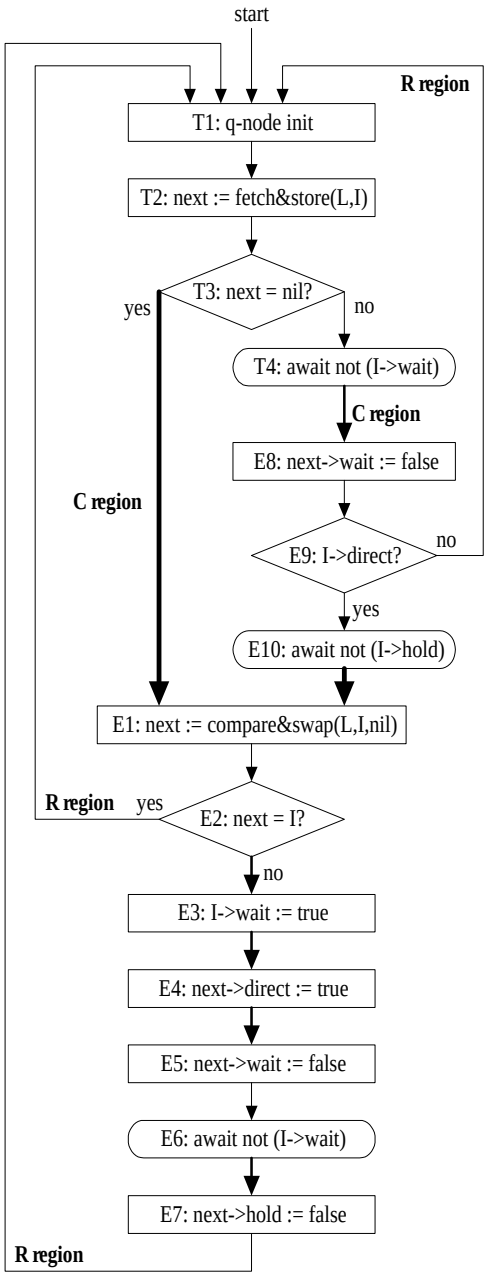

Figure 3: The base algorithm.

Here we explain how the algorithm works. Let *P* be the process that is executing. T1 is the initialization for q-node. T2 is to contend for the privileged value *nil* at the RMW register L. If *P* is the first to execute the *fetch&store*, it will have the test result ``yes'' later at T3 and enter C region immediately. Meanwhile, the unique address of the *P* 's q-node (I) is stored at the RMW register as part of the atomic operation at T2. For those contending processes that have been defeated at T2, each will be busy waiting at T4 by a local spin on the *wait* bit of its q-node. The winning process *P* will be charged some controller duty. The popular *compare&swap* primitive, see Figure 1, is used by the current controller to decide whether there are other processes interested in entering critical section. If not, the *nil* value is assigned to L and the controller has nothing else to do. If so, the value of L is returned and assigned to the private variable *next*. That value is the address of some process's q-node and that process will be assigned as the next controller. The two other bits (*direct* and *hold*) of each q-node are used to transfer the duty of controller from the current one to the next. The *direct* bit is to inform the next controller that it is assigned as such. The *hold* bit is to hold the next controller at E10 until the current controller finishes a run of wake-up operations and executes E7. It is easy to observe from the flowchart that a process will be able to leave E region if it passes E6. A process cannot be kept waiting indefinitely at E6 because the *fetch&store* primitive regulates the q-node addresses in such a way that the wake-up signal sent by *P* at E5 is bound to come back in finite steps as a signal to release *P* at E6.

Discussion

The base algorithm decented from work by Fu et al. [3]. The full-fledged algorithm reused the well-known idea of *k*-ary tournament tree and succeeded in keeping hot spot contention constant *k*. Prior solutions require that *k* be two, while we allow an arbitrary constant *k*. Hence, our solution reveals the existence of a tight bound for the trade-off that matches the intuition of

flexibility in the trading.

Like all known tournament tree-based mutual exclusion algorithms, the full-fledged algorithm does not enjoy bounded-bypass for fairness. It enjoys lockout freedom, though. The difficulty here is that although we do not allow zero-speed process, we do allow high-speed processes to take steps with no upper limit. Those high-speed processes can ruin any preset bound for bypass, if any kind of tournament tree is used in order to reach optimality in the trade-off.

四、計畫成果自評

本計畫大致上沿 Dwork、Herlihy 與 Waarts 在 1997 年 Journal of the ACM 的文章[1]之方向，提出較具實用性調節功能之改良，但不失其原有學理性之最佳化之特性。自評在 Formalism 程度稍弱，演算法尚缺正確性之嚴謹證明；在實用性方面稍有改良。

五、參考文獻

[1] Cynthia Dwork, Maurice Herlihy and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6): 779-805, November 1997.

[2] Leslie Lamport. The Mutual exclusion problem – Part I and II. *Journal of the ACM,* 33(2): 313-348, April 1986.

[3] S. S. Fu and N.-F. Tzeng, A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems,* vol . 6, no. 6, pp. 628-639, June 1997.

[4] J. H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing,* vol. 9, pp. 51-60, Spring-Verlag 1995.

[5] Yih-Kuen Tsay. Deriving a scalable algorithm for mutual exclusion. *In Proceeding of the 12th Int. symposium on distributed computing (DISC'98),* Sep.1998.