Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-90-18

1990-05-01

Dining with Synchronized Forks

Authors: Jerome Plun and Gruia-Catalin Roman

The arguments against centralized solutions focus on the performance bottleneck associated with a single central uniprocessor having a limited throughput and, possibly, a small number of ports. These limitations can be overcome to a large extent if the central processor is replaced by a modem SIMD (Single Instruction Multiple Data) machine. Several order of magnitude gains in parallelism are thus achievable while maintaining the logical simplicity of a centralized control. We call such a scheme parallel synchronous control (PSC). In this paper, we explore this approach by presenting a PSC solution to the classical Dining Philosopher problem and by contrasting it with a centralized one in which the philosophers are serviced sequentially.

Follow this and additional works at: http://openscholarship.wustl.edu/cse_research

Recommended Citation

Plun, Jerome and Roman, Gruia-Catalin, "Dining with Synchronized Forks" Report Number: WUCS-90-18 (1990). *All Computer Science and Engineering Research.*
http://openscholarship.wustl.edu/cse_research/693

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

DINING WITH SYNCHRONIZED FORKS

Jerome Plun
Gruia-Catalin Roman

WUCS-90-18

May 1990

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

The arguments against centralized solutions focus on the performance bottleneck associated with a single central uniprocessor having a limited throughput and, possibly, a small number of ports. These limitations can be overcome to a large extent if the central processor is replaced by a modern SIMD (Single Instruction Multiple Data) machine. Several order of magnitude gains in parallelism are thus achievable while maintaining the logical simplicity of a centralized control. We call such a scheme parallel synchronous control (PSC). In this paper, we explore this approach by presenting a PSC solution to the classical Dining Philosophers problem and by contrasting it with a centralized one in which the philosophers are serviced sequentially.

Index terms:

Centralized control versus synchronous control

Deadlock

Dining Philosophers

Distributed systems

Mutual exclusion

SIMD processing

Synchrony

1. Introduction

The last decade has been dominated by a popular trend toward distributed computing and has been marked by much research on the development of algorithms that exhibit little or no centralized control. Dijkstra's original solution of the Dining Philosophers problem [4], for instance, relied on the use of semaphores—a construct that emerged in a multiprogramming environment where centralized control was a reasonable choice. By contrast all subsequent solutions attempted to cope with the challenges of a totally distributed control. Chang presented the first distributed solution to the problem. Lynch [5] addressed it by presenting a general solution to the static resource allocation problem. A randomized algorithm to solve the Dining Philosophers problem was proposed by Rabin and Lehmann [6]. While these algorithms use shared memory variables, Chandy and Misra [3] proposed a solution using the message passing model. In general, all these algorithms treat the philosophers as processes and the forks as shared data. Agha [2], and Aggarwal, Barbara and Meth [1], however, proposed solutions where both the philosophers and the forks are processes.

The arguments against centralized solutions focus on the performance bottleneck associated with a single central uniprocessor having a limited throughput and, possibly, a small number of ports. These limitations can be overcome to a large extent if the central processor is replaced by a modern SIMD (Single Instruction Multiple Data) machine. Several order of magnitude gains in parallelism are thus achievable while maintaining the logical simplicity of a centralized control. We call such a scheme parallel synchronous control (PSC). In this paper, we explore this approach by presenting a PSC solution to the classical Dining Philosophers problem and by contrasting it with a centralized one in which the philosophers are serviced sequentially.

The presentation is divided in four sections. Section 2 introduces the Dining Philosophers problem and gives a general view of the centralized solution. Section 3 describes the solution using centralized control in the form of an explicit manager of the forks. Section 4 presents the parallel version of the solution where each fork is "working" in synchrony with the others. Section 5 shows the absence of deadlock in the parallel synchronized version.

2 Problem definition

In the Dining Philosophers problem, N philosophers are gathered in a common area to think and, occasionally, to eat. To do this second activity, they have a table with N chairs and N forks, each fork being located between two chairs. When a philosopher is hungry, he chooses a seat, picks up the two forks located immediately to his left and right, and eats. Once done eating, he puts down the two forks and

resumes thinking. We denote each philosopher by P_i and a fork by F_j, P_i having F_{i-1} on his left and F_i on his right.

In general centralized solution shared variables are used to communicate between the philosophers and a server supervising the seating of the philosophers and the availability of the forks. Each philosopher is represented by a separate process which seeks permission to eat from the special entity. This arrangement is shown in Figure 1. The philosophers processes are "seated" around the server, depicted as a table holding the forks. Read and write accesses to the variables shared by the philosophers and the central server are represented by arrows. All read and write actions are considered atomic in the sense that whenever a read overlaps a write to the same variable , the value returned by the read is either the value before or after the write. The shared variables $Phil_i$ store the status of each philosopher, i.e., thinking, trying to eat, or done eating. $Left_i$ and $Right_i$ respectively inform the philosopher i that his left or right fork has been granted by the central server.

The algorithm used by the server is correct if it satisfies the following requirements: (1) mutual exclusion between any two neighbor philosophers, (2) absence of deadlock among the philosophers, and (3) fairness in the sense that any philosopher willing to eat will eventually do so. In the two following sections, we will examine two specific protocols employed by the server.

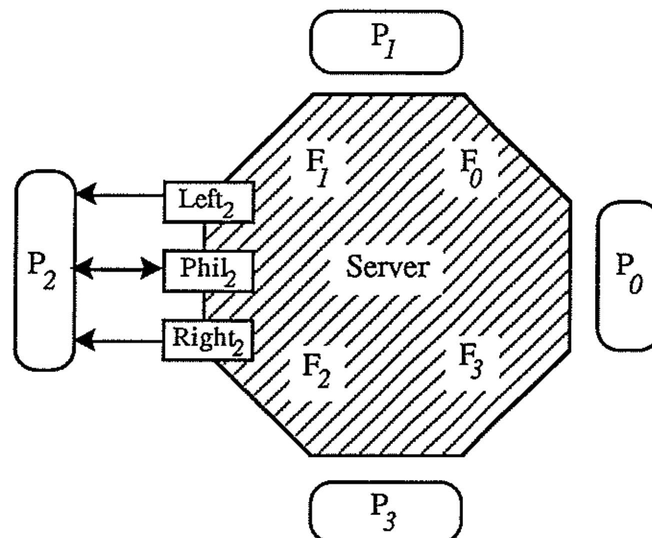

Figure 1: Seating of N=4 philosophers

3 Traditional centralized control

In this section, as depicted in Figure 1, the server is a single process that has complete knowledge of the status of each philosopher (communicated by the variables Phil$_i$) and grants forks based on this global knowledge of the situation. The server repeatedly serves each philosopher in sequential order, checking for a possible change of status of the corresponding philosopher or the availability of the forks related to that philosopher. Figure 2 contains the code for a philosopher process and the server. The server keeps track of granted forks using the variables $g(i,j)$ meaning "philosopher i has been granted fork j." Each computation is modulo N. Mutual exclusion is easily enforced by preventing a philosopher from using a fork if the other philosopher already has it. Deadlock is prevented by letting at most N-1 philosophers eat at any one time. This ensures that at least one philosopher can get his two forks. Finally any philosopher accepted in the waiting list W will eat after at most N-1 other philosophers thus guaranteeing the fairness of the algorithm.

4 Parallel synchronized control

In this section, the server is redefined based on the concept of parallel synchronous control. The single waiter is replaced by N synchronized fork processes, each responsible for the management of one fork. The reasoning leading to this solution is as following. Each philosopher only has to deal with two forks. Similarly, each fork is only concerned with the status of the two philosophers that could grab it. Thus we can break the management of the forks into N local fork processes, each accessing only 4 shared variables. This important reduction of accessed shared variables—compared with 3N accessed by the central "waiter"—combined with an increase in the number of ports (potentially one for each process) results in a much better throughput due to an increased parallelism of the I/O activities. Since no fork requires any information about the other forks, they can all compute in parallel. However, every pair of forks F_i and F_{i+1} needs the status of common philosopher, and the status as seen by each fork through access to the corresponding shared variable must be the same. This condition can be easily enforced by means of a global synchronization of the fork processes. In this context, however, we need to reexamine our definition of atomic read and write. A synchronous atomic read performed by several synchronized processes is an atomic read whose value is "broadcast" to all the processes involved. Similarly, a synchronous atomic write to a variable changes that variable—using an atomic write—to the common value written by all the processes involved. Based on these definitions, we need to synchronize every pair of forks to ensure that they read the same philosopher status, i.e., all the forks have to execute synchronously as it is the case in a typical SIMD machine. Figure 3 describes the relations between forks and philosophers by means of the shared variables.

Variables shared by Philosopher$_i$ and the waiter:

```
    Left_i, Right_i ∈ {True,False}: initially False;
    Phil_i ∈ {'think','hungry','done'}: initially 'think';
where i ∈ {0..N-1}
```

Code for Philosopher$_i$:

```
do forever
    think;
    Phil_i := 'hungry';
    do ¬Left_i or ¬Right_i → skip; od
    eat;
    Phil_i := 'done';
    do Phil_i = 'done' → skip; od
od
```

Code for the waiter:

```
Local variables:
    p ∈ {'think','hungry','done'};
    g(i,i-1), g(i,i) ∈ {True,False} : initially False;
    W[i] set of {0..N-1} : initially φ;
where i ∈ {0..N-1}

do forever
    do for i = 0..N-1
        p := Phil_i ;
        if  hungry(p) ∧ |W| < N ∧ i ∉ W
            →  W := W ∪ {i} ; fi
        if  hungry(p) ∧ i ∈ W ∧ free(i-1)
            →  g(i,i-1) := True ; fi
        if  hungry(p) ∧ g(i,i-1) ∧ free(i)
            →  g(i,i ) := True  ; fi
        if  done(p)
            →  g(i,i-1) := g(i,i ) := False; W := W - {i}; Phil_i := 'think'; fi
        Left_i , Right_i := g(i,i-1) , g(i,i) ;
    od
od
```

where $free(i) \equiv \overline{g(i\text{-}1,i)} \wedge \overline{g(i,i)}$, $hungry(p) \equiv (p=\text{'hungry'})$, $done(p) \equiv (p=\text{'done'})$

Figure 2: Centralized management of the forks

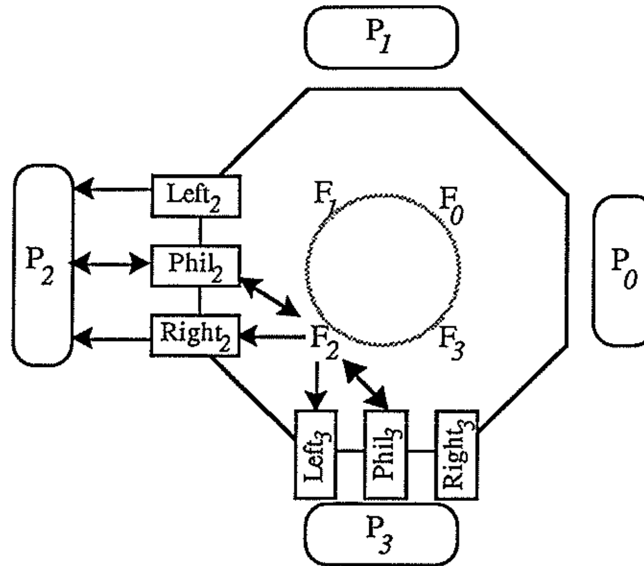

Figure 3: Relations between forks and philosophers

Each fork is in a state that reflects the status of both philosophers sharing the fork. The state is a tuple <*A*,*B*> where *A* and *B* can take the values 0, 1 or 2. These values reflect the "age" of a philosopher or, more precisely, how long he has been waiting for the corresponding fork while the other philosopher was eating. This scheme is used to prevent a philosopher from eating twice while his neighbor is waiting, thus enforcing fairness between each pair of philosophers. A value of 0 means that the corresponding philosopher is thinking. A value of 1 indicates that the philosopher has been granted the fork. Finally, if a philosopher wants a fork that is currently used by the other, the corresponding value is 2. The transitions between each potential state are conditioned by the change of status of the two philosophers. Figure 4 shows all the possible states of a fork and the conditions under which each transition is taken. The simplest case corresponds to an available fork F_i wanted by a single philosopher P_i: the fork state changes from <0,0> to <1,0>, showing that the fork has been granted to the philosopher, and then back to <0,0>, once the philosopher is done eating. In a more complex situation like two philosophers P_1 and P_2 wanting the fork F_1 simultaneously, the sequence of state can be <0,0> to <2,1> (P_2 gets the fork, P_1 has to wait), followed by ≤1,0> (P_2 is done, P_1 is then allowed to eat), and finally back to <0,0> (when P_1 is done). Figure 5 contains the code of a philosopher and a fork processes based on this state diagram. The multiple IF statement of each process is executed by each process according to the process local state while the other statements—which all involve shared variables—are performed simultaneously by all the forks. Thus, a fork process executing quickly its state transition has to wait for the other forks to complete their state

transitions before the synchronous reads and writes are executed in a lock-step fashion. Each synchronous statement is indicated in the algorithm by the symbol ♦.

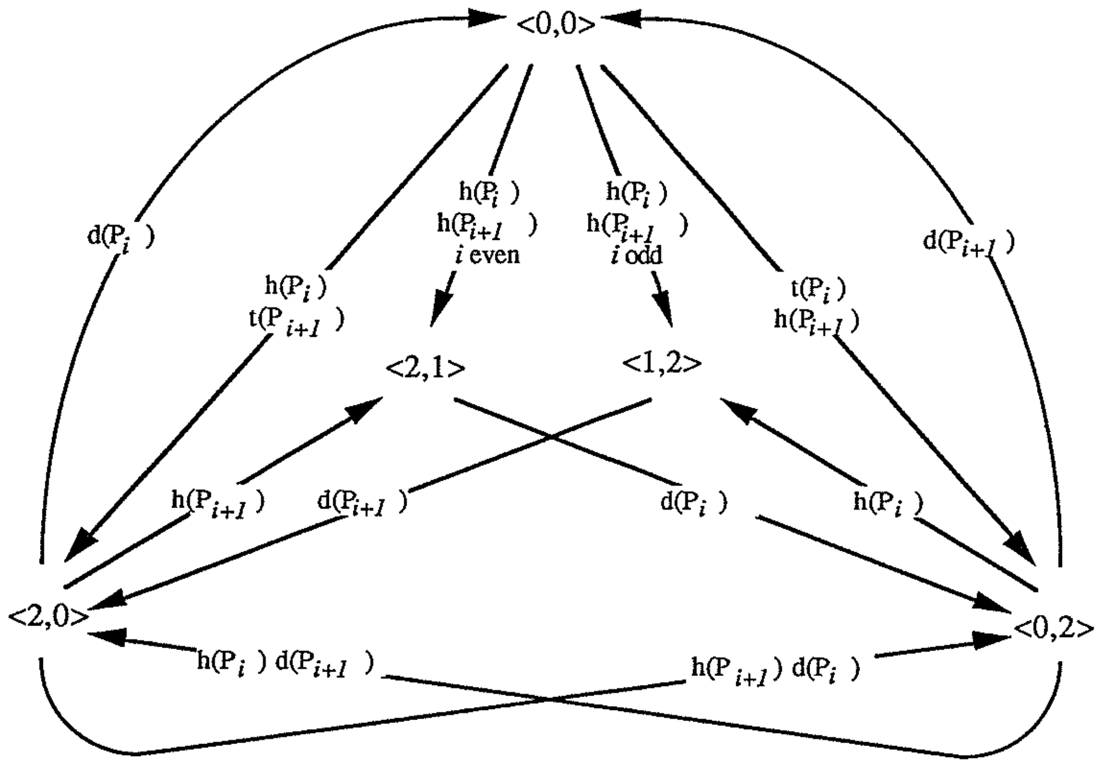

Figure 4: State of a fork i
where $h(P) \equiv (P=\text{'hungry'})$, $d(P) \equiv (P=\text{'done'})$ and $t(P) \equiv (P=\text{'think'})$

5 Proof outline

In this section, we first demonstrate that deadlock is avoided. We then show by example that, in the absence of synchronization, the algorithm can lead to deadlock.

Claim 1: The algorithm is deadlock-free if the forks are synchronized.

Variables shared by $Philosopher_i$ and $Fork_i$:

```
    Left_i, Right_i ∈ {True,False}: initially False;
    Phil_i ∈ {'think','hungry','done'}: initially 'think';
where i ∈ {0..N-1}
```

Code for $Philosopher_i$:

```
do forever
    think;
    Phil_i:= 'hungry';
    do ¬Right_i or ¬Left_i+1 → skip; od
    eat;
    Phil_i := 'done';
    do Phil_i = 'done' → skip; od
od
```

Code for $Fork_i$:

```
Local variables:
    P_i, P_i+1 ∈ {'think','hungry','done'};
    L_i, R_i+1 ∈ {0,1,2} : initially 0;

do forever
  ▶ P_i , P_i+1 := Phil_i , Phil_i+1 ;
    if L_i = 0 , R_i+1 = 0 , t(P_i) , h(P_i+1)              → R_i+1:=1;
    ▯ L_i = 0 , R_i+1 = 0 , h(P_i) , t(P_i+1)              → L_i := 1;
    ▯ L_i = 0 , R_i+1 = 0 , h(P_i) , h(P_i+1) , i even     → L_i:=1; R_i+1:=2;
    ▯ L_i = 0 , R_i+1 = 0 , h(P_i) , h(P_i+1) , i odd      → L_i:=2; R_i+1:=1;
    ▯ L_i = 0 , R_i+1 = 1, t(P_i) , d(P_i+1)               → R_i+1:=0; P_i+1:='think';
    ▯ L_i = 0 , R_i+1 = 1 , h(P_i) , h(P_i+1)              → L_i:=2;
    ▯ L_i = 0 , R_i+1 = 1 , h(P_i) , d(P_i+1)              → L_i:=1; R_i+1:=0; P_i+1:='think';
    ▯ L_i = 1 , R_i+1 = 0 , t(P_i+1) , d(P_i)              → L_i:=0; P_i:='think';
    ▯ L_i = 1 , R_i+1 = 0 , h(P_i+1) , h(P_i)              → R_i+1:=2;
    ▯ L_i = 1 , R_i+1 = 0 , h(P_i+1) , d(P_i)              → L_i:=0; R_i+1:=1; P_i:='think';
    ▯ L_i = 2 , R_i+1 = 1 , d(P_i+1)                       → L_i:=1; R_i+1:=0; P_i+1:='think';
    ▯ L_i = 1 , R_i+1 = 2 , d(P_i)                         → L_i:=0; R_i+1:=1; P_i:='think';
    fi
  ▶ Phil_i , Phil_i+1 := P_i , P_i+1 ;
  ▶ if L_i = 1        → Right_i :=True; fi
  ▶ if R_i+1 = 1      → Left_i+1 := True; fi
od
```

where h(P) ≡ (P='hungry') , d(P) ≡ (P='done') and t(P) ≡ (P='think');

Figure 5: Synchronized forks

Proof:

The philosophers will be deadlocked if each "owns" one fork but is not able to get the other. This happens if all forks are in state <2,1> (or <1,2>). To prove that this situation can not occur, we need to show that none of the non-deadlocked states (thereafter referred as eligible states) can lead to a deadlocked global state. The global state is defined as the combination of the states of each fork at any step.

We only need to look at the eligible states from which a deadlock can be reached in one step. Without loss of generality, we restrict our attention to the deadlocked state in which all forks are in the state <2,1>. Based on the state transition graph, one can see that if a fork is in state <2,1>, its previous state was either <0,0>, <0,1> or <2,1>. And, since a philosopher is seen by two forks, an "age" of 0 for a fork *i*—meaning that the philosopher is not present at the table—implies a corresponding "age" of 0 for the fork *i+1*. With those constraints, the possible states of the set of forks {*i–1,i,i+1*} potentially leading in one step to the deadlocked state {<2,1>,<2,1>,<2,1>} are limited to 4. And, as shown in the table 6, none can actually lead to this deadlocked state. We can conclude that the algorithm is deadlock-free since a deadlock state can not be reached from an eligible state. ∎

{I–1,I,I+1}	Conditions required to move to <2,1>,<2,1>,<2,1>
<0,0>,<0,0>,<0,0>	I–1 even, I even, I+1 even
<0,0>,<0,0>,<0,1>	I–1 even, I even
<0,0>,<0,1>,<2,1>	only potential state of forks I–2,I–1,I is <0,0>,<0,0>,<0,1> which can't lead to <2,1>,<2,1>,<2,1> as shown above.
<0,1>,<2,1>,<2,1>	same as previously with forks I–1,I–2,I–3

Table 6: No state leads to a deadlocked state

Claim 2: The algorithm is not deadlock-free if the forks are not synchronized.

Proof:

The previous proof is based on the fact that all forks check the philosophers' status at once, and change state synchronously. Let us assume that the forks are not synchronized. In the worst case, this means that only one fork is selected and can change state at a time. Also no fork is required to notice the fact that a philosopher has entered the room. With this scheme, it is possible to devise a sequence of steps that leads to a deadlocked global state. The table 7 shows one such sequence with 4 forks and 4 philosophers. The state of each fork is given after each action (which corresponds to one or more steps). ∎

Actions	States				Explanation
	F_1	F_2	F_3	F_4	
initially	<0,0>	<0,0>	<0,0>	<0,0>	¬P(i), i=1...4
P_2 & P_4 enter	<0,0>	<0,0>	<0,0>	<0,0>	P(2), P(4)
F_1 checks	<1,0>	<0,0>	<0,0>	<0,0>	P(4), ¬P(1) -> <1,0>
F_3 checks	<1,0>	<0,0>	<1,0>	<0,0>	P(2), ¬P(3) -> <1,0>
P_1 & P_3 enter	<1,0>	<0,0>	<1,0>	<0,0>	P(1), P(3)
F_1 checks	<1,2>	<0,0>	<1,0>	<0,0>	<1,0>, P(1) -> <1,2>
F_2 checks	<1,2>	<1,2>	<1,0>	<0,0>	P(1), P(2), 2 even -> <1,2>
F_3 checks	<1,2>	<1,2>	<1,2>	<0,0>	<1,0>, P(3) -> <1,2>
F_4 checks	<1,2>	<1,2>	<1,2>	<1,2>	P(3), P(4), 4 even -> <1,2>

Table 7: Deadlock reached without synchrony among forks

6 Conclusion

Rapid technological changes force the computer scientist continuously to reexamine the question of what is a reasonable solution for a particular problem or class of problems. In this paper we have shown the advent of very large SIMD machines adds a new nuance to the dichotomy between centralized and distributed control: the parallel synchronous control (PSC). While essentially centralized, PSC exhibits very high levels of parallelism and a strong potential for reliability. Although PSC may not scale up to the same degree as a fully distributed control, it appears to be a reasonable middle ground option that has not been previously explored. Our solution to the dining philosophers problem illustrates one application of this concept.

7 References

1. S. Aggarwal, D. Barbara and K.L. Meth. A software environment for the specification and analysis of problems of coordination and concurrency. IEEE Transactions on Software Engineering, Vol. 14, No 3, March 1988.

2. G Agha. ACTORS: a model of concurrent computation in distributed systems. M.I.T. Press, 1986.

3. K. Chandy and J. Misra. The drinking philosophers problem. ACM Transactions on Programming Languages and Systems, 6(4):632-646, October 1984.

4. E.W. Dijkstra. Hierarchical ordering of sequential processes. Acta Informatica, 115-138, 1971.

5. N. Lynch. Upper bounds for static resource allocation in a distributed system. Journal of Computer and Systems Sciences, 23(2):254-278, October 1981.

6. M.O. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In Proceedings of 8th ACM Symposium on Principles of Programming Languages, pages 133-138, 1981.