Report Number: WUCS-91-32

1991-05-30

Parallel Synchronous Control

Gruia-Catalin Roman and Jerome Y. Plum

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 some extent if the central processor is replaced by a modern SIMD (Single Instruction Multiple Data) machine. Several orders 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. The PSC solution... Read complete abstract on page 2.

Parallel Synchronous Control

Gruia-Catalin Roman and Jerome Y. Plum

Complete 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 some extent if the central processor is replaced by a modern SIMD (Single Instruction Multiple Data) machine. Several orders 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. The PSC solution is derived directly from a fair centralized solution without needing reverification.

Parallel Synchronous Control

Gruia-Catalin Roman
Jerome Y. Plun

WUCS-91-32

30 May 1991

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 some extent if the central processor is replaced by a modern SIMD (Single Instruction Multiple Data) machine. Several orders 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. The PSC solution is derived directly from a fair centralized solution without needing reverification.

Keywords: concurrency, algorithms, synchronization, architecture.

Correspondence: All communications regarding this paper should be addressed to

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

(314) 889-6190
roman@CS.WUSTL.edu
fax: (314) 726-7302

1. Introduction

The last decade has been dominated by a popular trend towards distributed computing and has been marked by much research on the development of algorithms that exhibit little or no centralized control. The arguments against centralized solutions focused originally on the performance bottleneck associated with a single central uniprocessor having a limited throughput and, possibly, a small number of ports. Although it is generally accepted that centralized solutions are not likely to scale up, current system architectures offer new opportunities for exploiting centralized control strategies for multiprocessor structures and local area networks. For instance, the market success of CSMA/CD (Carrier Sense Multiple Access with Collision Detection) LAN's [14], such as Ethernet [17], encouraged the development of distributed algorithms that employ broadcast as a communication primitive [16]; data parallel algorithms [13] take advantage of the global synchronous control of modern SIMD (Single Instruction Multiple Data) machines; and barrier synchronization schemes are being pursued by a number of authors [5, 12].

In this paper we explore yet another centralized control strategy. It is called *parallel synchronous control* (PSC) and is based on the simple notion of substituting an SIMD machine in place of a uniprocessor running a centralized control protocol. The result is a major increase in both the throughput and the number of input/output ports available to the controller. Gains of several orders of magnitude in parallelism are thus achievable while maintaining the logical simplicity of a centralized control. Possible uses of PSC include distributed simulation and multimedia. In a distributed simulation, multiple local copies of the global simulation time may be updated simultaneously by employing some synchronous algorithm (e.g., tree structured computation of the minimum time over all scheduled events) or built-in global operations (e.g., on a Connection Machine, counting the number of active processors using the *count* operation and incrementing the local times when the number is zero). In a multimedia environment multiple devices (e.g., displays, projection systems, sound sources, robots, animations, etc.) must be coordinated in a synchronous fashion. PSC can simplify the development of the control software and the design of device interfaces.

For reasons of brevity and simplicity we choose the *dining philosophers problem* to illustrate the PSC idea. Dijkstra's original solution of the dining philosophers problem [11] relied on the use of semaphores—a construct that emerged in a multiprogramming environment where centralized control was a reasonable choice. All subsequent solutions attempted to cope with the challenges of a totally distributed control. Lynch [15] addressed this problem by seeking a general solution to the static resource allocation problem. A randomized algorithm to solve the dining philosophers problem was proposed by Rabin and Lehmann [18]. While these algorithms use shared memory, Chandy and Misra [7] 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 [3], and Aggarwal, Barbara and Meth [2], however, proposed solutions where both the philosophers and the forks are processes. Our own PSC-type solution treats each fork as a process running on a distinct processor of a SIMD machine while the philosophers execute asynchronously on a MIMD (Multiple Instruction Multiple Data) machine. Each philosopher communicates with its neighboring forks via a small amount of shared memory.

The remainder of the paper is organized into four parts. Section 2 introduces the notation system used in this paper. The notation is borrowed from a model of concurrency called Swarm, a model which allows one to specify and reason about computations which combine synchronous and asynchronous modes of execution. Section 3 presents a fair centralized solution to the dining philosophers problem and outlines its correctness proof. Section 4 revisits the solution and distributes it across the processors of a SIMD machine. The simulation of the resulting algorithm and architecture on a hypercube are also discussed. A brief summary and conclusions are given in Section 5.

2. Swarm

Swarm [20] belongs to a class of languages and models that use tuple-based communication and is the first of its kind to have an assertional style proof logic [9, 10]. Other languages and models in this class are Linda [6], Associons [19], GAMMA [4]. Of particular import for this paper is the ease with which Swarm can accommodate a variety of programming paradigms (e.g., shared variables, message passing, and rule-based programming) and architectures (e.g., synchronous, asynchronous, reconfigurable, etc.).

Dataspace representation. In Swarm, the entire computation state is captured by a set of tuple-like entities called the dataspace. The dataspace consists of three kinds of entities: data tuples (capturing the data state), transactions (identifying actions whose execution can change the program state), and synchrony relation entries (which will be completely ignored until the end of this section). Transactions and data tuples assume the form:

- data_type_name(sequence_of_values)—e.g., A(2,66) may be the tuple representation of an array entry;
- transaction_type_name(sequence_of_values)—e.g., Swap(2) may be a transaction.

Primitive operations. The dataspace is subject to query, deletion, and insertion of the entities it contains. To query for the existence of an entity in the dataspace one simply treats the tuple description as a predicate over the dataspace. Insertions are specified by fully instantiated tuples and deletions are specified by tagging a fully instantiated tuple with a dagger (†). Any type of entity can be queried, inserted, or deleted, except transactions which, for technical reasons can not be explicitly deleted. Here are some examples of queries and actions one can specify in Swarm:

- Swap(2)—(as a query) checks if this transaction is in the dataspace;
- $[\forall\ i : 1 \leq i < 8 :: A(i,0)]$—(only as a query) checks if the array A contains only zeros for index values 1 through 8;
- Swap(2)—(as an action) inserts this transaction into the dataspace;
- A(2,5)†—(as an action) deletes this array entry from the dataspace;

Atomic transformations. Both computation and communication are expressed as atomic transformations of the dataspace. A simple atomic transformation is defined in terms of a query followed by an action list consisting of deletions and insertions. For instance

$$i,j,x,y : i < j \wedge A(i,x) \wedge A(j,y) \wedge x > y \rightarrow A(i,x)\dagger,\ A(j,y)\dagger,\ A(i,y),\ A(j,x)$$

states that two unsorted array entries are subjected to an exchange by deleting the old instances and inserting new ones. The query, similar to a Prolog goal, is an arbitrary predicate which may involve testing for the presence or absence of data-tuples, transactions, and synchrony relation entries in the dataspace. A successful query binds the variables listed before the query (existentially quantified by implication) to values used to compute the dataspace deletions and insertions. Deletions always precede insertions. If the query evaluates to false, no deletions or insertions are performed.

Commas may be used inside the query as shorthand for the logical *and* (∧) and the order in which deletions and insertions are listed is immaterial. Actually, when the items being deleted are present in the query part, their deletion can be marked by daggers inside the query (only as a shorthand notation). Using these conventions the transformation above becomes

$$i,j,x,y : i < j,\ A(i,x)\dagger,\ A(j,y)\dagger,\ x > y \rightarrow A(i,y),\ A(j,x)$$

Static composition. The ||-operator, as in UNITY [8], may be used to combine syntactically several transformations—the resulting transformation may not be equivalent to any serial execution of the component transformations since all queries precede all deletions which, in turn, precede all insertions. For instance, a transformation which swaps synchronously all unsorted odd-even positions of an array of size *N* can be stated as

```
[ || i : $1 \le i \le N$ , i mod 2 = 1 ::
    x,y : $A(i,x)\dagger, A(i+1,y)\dagger, x > y \rightarrow A(i,y), A(i+1,x)$
]
```

The construct used here is called a generator and it is widely used in Swarm to define groups of objects, statements, or dataspace entities.

Naming. The composite transformation above can be parameterized and can be given a name:

```
Swap(N) ≡
    [ || i : 1≤i≤N , i mod 2 = 1 ::
        x,y : A(i,x)†, A(i+1,y)†, x > y → A(i,y), A(i+1,x)
    ]
```

It thus becomes a transaction type definition. An instance of *Swap(N)*, such as *Swap(4)*, is called a transaction. The component transformations are called subtransactions.

Transaction execution. The set of transactions present in the dataspace define the control state of the program. The execution of a Swarm program assumes that each transaction in the dataspace is eventually selected and the transformation specified by its type definition is executed atomically. By convention, a transaction is always deleted after its execution (unless explicitly reinserted) and no transaction is allowed to delete any other transactions.

Dynamic composition. At run time, independently created transactions may be coupled to form *synchronic groups*, disjoint sets of transactions which are executed synchronously. The result is a dynamic form of the ||-operator with the semantics of executing a synchronic group being essentially the same as that of a group of subtransactions which are part of the same transaction. The execution rule is changed by requiring that whenever a transaction is selected for execution, the entire synchronic group to which it belongs is executed synchronously and deleted implicitly, unless the members of the group recreate themselves explicitly.

In order to specify the structure of the synchronic groups, the dataspace includes synchrony relation entries which assume the form

$$\text{transaction\_type\_name}_1(\text{sequence\_of\_values}_1) \sim \text{transaction\_type\_name}_2(\text{sequence\_of\_values}_2)$$

and which may be queried, inserted, and deleted like any other dataspace entities. For instance,

$[\ i : 2 \leq i < 8 \wedge \text{even}(i) :: \text{Swap}(i) \sim \text{Swap}(i+2)\ ]$

requires that henceforth the transactions *Swap(2)*, *Swap(4)*, *Swap(6)*, and *Swap(8)* be executed synchronously whenever two or more are present in the dataspace. Similarly,

$[\ i : 2 \leq i < 8 \wedge \text{even}(i) :: (\text{Swap}(i) \sim \text{Swap}(i+2))\dagger\ ]$

removes the synchronous execution requirement among the four transactions involved.

The reflexive transitive closure of the synchrony relation, denoted by "≈", partitions the universe of transactions into disjoint sets whose intersections with the dataspace define the currently active synchronic groups. The notation *(Swap(i) ≈ Swap(j))* allows one to query the closure of the synchrony relation. The synchrony relation allows one to specify explicitly the synchronous execution of the components of some computation. For instance, to specify an SIMD configuration, each sequential process is described by a set of transactions, where each transaction creates a single continuation, and all the transactions are part of a single synchronic group. The importance of the synchrony relation will become apparent in Section 4 where the PSC structure is introduced.

3. Fair Centralized Solution

The starting point for the PSC solution is a fair centralized algorithm involving *N* philosophers—numbered from 0 to *N*-1—and one waiter that manages the forks. (The algorithm is given in Figure 2.) Each philosopher has a cyclic behavior: after thinking for a while, he becomes hungry and requests the two forks he needs to eat, he eats once the forks have been granted, and upon completion of his "meal" he returns the forks and resumes thinking. Each philosopher starts in a thinking phase. Because Swarm lacks any sequential constructs, the three phases are represented by the distinct transactions *Thinking(i)*, *Waiting(i)*, and *Eating(i)*. Exactly one of the three is always present in the dataspace for each value of *i*. Communication between each philosopher and the waiter is performed using four shared variables encoded as tuples. When philosopher *i* becomes hungry, the tuple *hungry(i)* is inserted in the dataspace by the transaction *Thinking(i)* whose continuation is the transaction *Waiting(i)*. *Waiting(i)* continuously tests whether or not the needed forks have been granted. This is done by checking for the presence in the dataspace of the tuples *fork(α,(i-1) mod N,E)* and *fork(E,i,β)*—where the tag E indicates that the philosopher *i* is allowed to eat with that particular fork while α and β hold information regarding the philosophers located on the other side of each of the two forks. Once both forks have been granted, the transaction *Waiting(i)* is replaced by *Eating(i)* in which the actual processing is symbolized by the subtransaction *work_to_do → work , Eating(i).* Finally, once the "meal" is finished, *Eating(i)* inserts the tuple *done(i)* to inform the waiter that the granted forks were returned, but cannot make a new request until

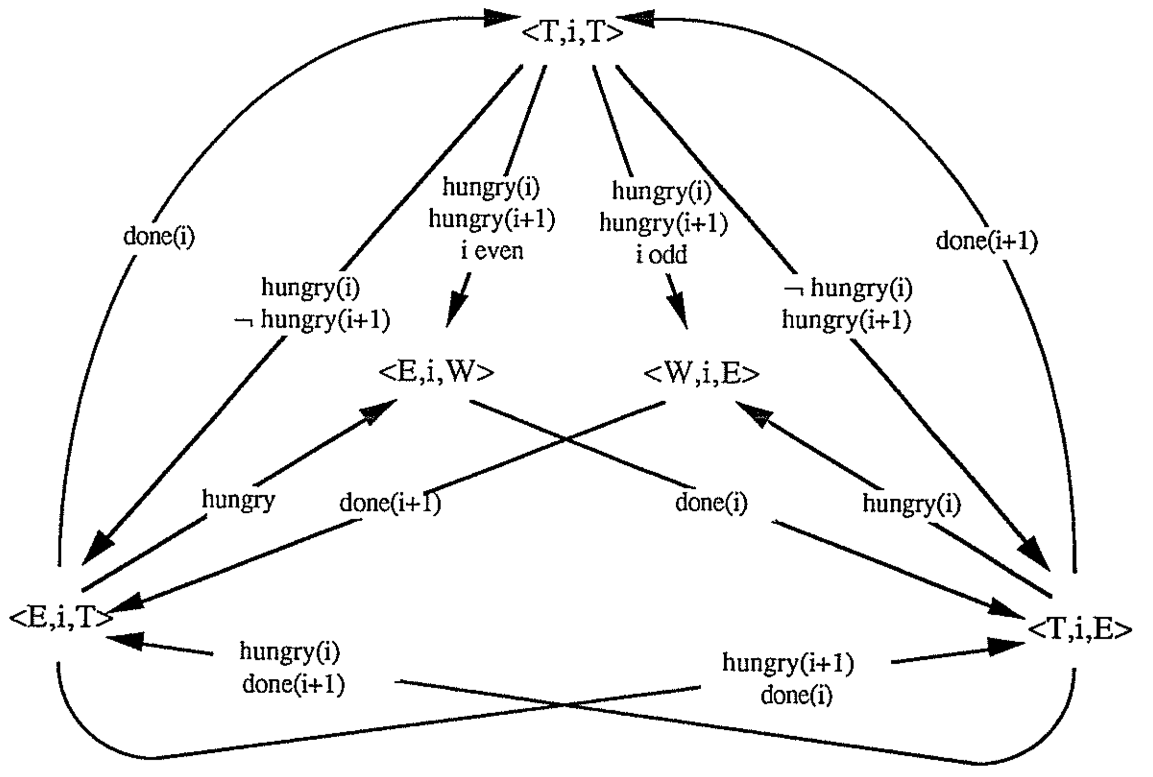

Figure 1: State Diagram for Fork *i*

the waiter signals that the meal completion has been taken into account by removing the tuple *done(i).*

The waiter embodies the fork granting policy which is designed to be fair in the sense that no hungry philosopher is bypassed by its neighbors more then once. Figure 1 captures the policy as a finite state diagram which is encoded in Swarm as the transaction *Waiter* of Figure 2. The waiter looks for hungry and done philosophers and updates the status of the forks by removing and inserting *fork* tuples.

```
program Dining_Philosophers_with_Central_Waiter(N)
tuple types
    [ ι, λ, ρ : 0 ≤ ι < N, λ ∈ {T,W,E}, ρ ∈ {T,W,E} :: hungry(ι), fork(λ,ι,ρ), done(ι)]
transaction types
    Waiter ≡
        True → Waiter
        ‖ [ ‖ ι: 0 ≤ ι < N ::
                fork(T,ι,T)†,  ¬hungry(ι),  hungry((ι+1) mod N)†             → fork(T,ι,E)
            ‖   fork(T,ι,T)†,  hungry(ι)†,  ¬hungry((ι+1) mod N)             → fork(E,ι,T)
            ‖   fork(T,ι,T)†,  hungry(ι)†,  hungry((ι+1) mod N)†, even(ι)    → fork(E,ι,W)
            ‖   fork(T,ι,T)†,  hungry(ι)†,  hungry((ι+1) mod N)†, odd(ι)     → fork(W,ι,E)
            ‖   fork(T,ι,E)†,  ¬hungry(ι),  done((ι+1) mod N)†               → fork(T,ι,T)
            ‖   fork(E,ι,T)†,  done(ι)†,    ¬hungry((ι+1) mod N)             → fork(T,ι,T)
            ‖   fork(T,ι,E)†,  hungry(ι)†,  ¬done((ι+1) mod N)               → fork(W,ι,E)
            ‖   fork(E,ι,T)†,  ¬done(ι),    hungry((ι+1) mod N)†             → fork(E,ι,W)
            ‖   fork(T,ι,E)†,  hungry(ι)†,  done((ι+1) mod N)†               → fork(E,ι,T)
            ‖   fork(E,ι,T)†,  done(ι)†,    hungry((ι+1) mod N)†             → fork(T,ι,E)
            ‖   fork(W,ι,E)†,               done((ι+1) mod N)†               → fork(E,ι,T)
            ‖   fork(E,ι,W)†,  done(ι)†                                      → fork(T,ι,E)
        ];

    [ i : 0 ≤ i < N ::
    Thinking(i)  ≡  user_cond, ¬done(i) → Waiting(i), hungry(i)
                 ‖  otherwise → Thinking(i);

    Waiting(i)   ≡  α, β : fork(α,(i-1) mod N,E), fork(E,i,β) → Eating(i)
                 ‖  otherwise → Waiting(i);

    Eating(i)    ≡  work_to_do → work, Eating(i)
                 ‖  otherwise → Thinking(i), done(i)
    ]
initially
    [   Waiter,
        [ ι : 0 ≤ ι < N :: Thinking(ι), fork(T,ι,T)]
    ]
end
```

Figure 2: Centralized waiter controlling the forks

Each of these tuples contains the status of the two philosophers related to the fork, as perceived by that fork. The information pertaining to the left and right philosophers is stored respectively in the first and third components of the *fork* tuple. Since each philosopher starts in a thinking state, all forks have initially the symbol "T" in their 1st and 3rd components. When a requested fork is granted, the fork tuple is altered by placing an "E" in the appropriate component. If a philosopher requests a fork already granted, a "W" is placed in the related component to keep track of this pending request. Once a *done(i)* tuple is present in the dataspace—indicating that the corresponding philosopher is done eating—the fork is either granted to the other philosopher if he is waiting for it, or returns to its initial state. In both cases, the removal of the *done(i)* tuple allows the corresponding philosopher to try to eat again if he is willing to do so.

To prove that our algorithm is correct, we need to show that 1) no philosopher will starve, and 2) deadlock can not occur. In the following proofs, we use the notation *<left-right>* as an abbreviation for *fork(left,i,right)*.

Theorem 1: In the absence of deadlock, the algorithm in Figure 2 is starvation-free.

Proof: Starvation occurs when a philosopher can not succeed to gain control of both forks while his neighbors can. Whenever a philosopher is willing to eat but can not be granted a fork, the fork records the refusal in its state and, as soon as the other philosopher is done eating, the fork is granted to the waiting philosopher. Since once a fork is granted, it is kept by the philosopher until he has eaten, and since deadlock can not occur, a philosopher willing to eat has to wait at most for his two neighbors to eat once.

■

Theorem 2: The algorithm in Figure 2 is deadlock-free.

Proof: Let's assume that deadlock occurs. This means that each philosopher owns a fork and is waiting for the other, i.e., that all the forks are in state <E-W> (or all in state <W-E>). Let's say, without loss of generality, that the common state is <E-W>. Let T_{dead} be the last step where some forks had a state transition to <E-W> from a different state. Based on the fork state diagram, the source state of the transition must have been <T-T> or <E-T>.

a) Fork *i* in state <T-T> before T_{dead}:

If the fork *i+1* is also in state <T-T>, having both forks *i* and *i+1* change to <E-W> requires that both *i* and *i+1* be even, which is not possible. Hence, fork *i+1* must have been in a different state. Since fork *i* was in state <T-T>, philosophers *i* and *i+1* were still thinking. For fork *i+1* to not be in state <T-T>, philosopher *i+2* must be waiting for his forks (he can not be eating already since deadlock is supposed to occur after the next transition.) Hence, the state of fork *i+1* is <T-E>. But it is not possible for this fork to reach state <E-W> without philosopher *i+2* eating. So deadlock can not occur.

b) Fork *i* in state <E-T> before T_{dead}:

This requires that fork *i+1* be either in state <T-T> or <T-E>. As explained in the previous case, this can not lead to deadlock. ■

Theorem 3: The algorithm in Figure 2 would not be deadlock-free if the forks were 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, and that no fork is required to notice the fact that a philosopher became hungry. With this scheme, it is possible to devise a sequence of steps that leads to a deadlocked global state. The Table 1 shows one such sequence with four forks (F_1 through F_4) and four philosophers (P_1 through P_4). The state of each fork is given after each action (which corresponds to one or more steps). Any change in a fork's state is indicated with an underline. The explanations for the change are given in the right-most column. ■

Agent	States				Explanation
	F_1	F_2	F_3	F_4	
initially	<T-T>	<T-T>	<T-T>	<T-T>	$\neg$hungry(1) $\wedge$ $\neg$hungry(2) $\wedge$ $\neg$hungry(3) $\wedge$ $\neg$hungry(4)
P_2 & P_4	<T-T>	<T-T>	<T-T>	<T-T>	$\Rightarrow$ hungry(2) $\wedge$ hungry(4)
F_1	<E-T>	<T-T>	<T-T>	<T-T>	hungry(4) $\wedge$ $\neg$hungry(1) $\Rightarrow$ <E-T>
F_3	<E-T>	<T-T>	<E-T>	<T-T>	hungry(2) $\wedge$ $\neg$hungry(3) $\Rightarrow$ <E-T>
P_1 & P_3	<E-T>	<T-T>	<E-T>	<T-T>	$\Rightarrow$ hungry(1) $\wedge$ hungry(3)
F_1	<E-W>	<T-T>	<E-T>	<T-T>	<E-T> $\wedge$ hungry(1) $\Rightarrow$ <E-W>
F_2	<E-W>	<E-W>	<E-T>	<T-T>	hungry(1) $\wedge$ hungry(2) $\wedge$ 2 even$\Rightarrow$<E-W>
F_3	<E-W>	<E-W>	<E-W>	<T-T>	<E-T> $\wedge$ hungry(3) $\Rightarrow$ <E-W>
F_4	<E-W>	<E-W>	<E-W>	<E-W>	hungry(3) $\wedge$ hungry(4) $\wedge$ 4 even$\Rightarrow$<E-W>

Table 1: Deadlock reached without synchrony among forks

4. Distribution of the forks within an SIMD machine

In the previous section, we proved deadlock-freedom given that all forks change state based on the same view of the dataspace. This is enforced by the atomic nature of transaction execution in the Swarm model and the fact that the queries for all subtransactions are evaluated before the dataspace is modified through the removal of tuples and insertions of transactions and tuples. In Swarm there is no semantic difference between executing a transaction and a synchronic group. This allows us to partition the subtransactions of a single transaction across several new transactions which are part of the same synchronic group and exist in the dataspace whenever the original transaction existed. In other words, the transaction

$$T \equiv S_0 \parallel \ldots \parallel S_n$$

may be replaced by the transactions

$$T_1 \equiv S_0 \parallel \ldots \parallel S_{k_1}$$

$$T_2 \equiv S_{k_1+1} \parallel \ldots \parallel S_{k_2}$$

$$\cdots$$

$$T_m \equiv S_{k_{m-1}+1} \parallel \ldots \parallel S_n$$

without reverification of the program if the synchrony relation is modified to include the entries

$$[\ i : 1 \le i < m :: T_i \sim T_{i+1}\ ].$$

To make the transition from the single transaction *Waiter* of Figure 2 to a set of synchronously executing transactions, each allocated to a distinct processor of an SIMD machine, we simply need to partition the subtransactions of *Waiter* in some convenient way. We chose to break the *Waiter* transaction into *N* identical transactions *Waiter(i)*, each handling fork *i*. This was suggested by the locality of the shared data involved in the state change of a fork, namely only the corresponding *fork* tuple and the *hungry* and *done* tuples from the two philosophers able to request the fork. This scheme reduces the number of processors accessing a common shared memory from *N*+1 to 3, which reduces memory contention. The resulting algorithm is given in Figure 3. An ideal implementation of the algorithm would have an SIMD machine handling the forks by having each fork allocated to a different node, while philosophers would reside on external asynchronous processors—such as an MIMD machine or a network of independent machines.

Not having an SIMD computer available, we implemented this algorithm on a 64-nodes hypercube Ncube-7 [1]. The NCUBE is an MIMD machine with message-passing communication from single source to single destination. The shared memory accesses had to be replaced by request, grant and return messages between philosopher nodes (each philosopher on a different node) and the fork handler nodes (each handler on a different node). Instead of having tuples read by multiple transactions, several identical messages were sent to the appropriate nodes. To ensure that all fork nodes used only messages received by all their recipients, a barrier synchronization mechanism was used as follows: 1) each fork node read its pending messages; 2) every pair of forks "sharing" a philosopher exchanged their list of received messages hence

```
program Dining_Philosophers_with_Choosing_Forks(N)
    tuple types
        [ ι, λ, ρ : 0 ≤ ι < N, λ ∈ {T,W,E}, ρ ∈ {T,W,E} :: hungry(ι), fork(λ,ι,ρ), done(ι)]
    transaction types
        [ i : 0 ≤ i < N ::
        Waiter(i) ≡ True → Waiter(i)
                ‖  fork(T,i,T)†,  ¬hungry(i),  hungry((i+1) mod N)†             → fork(T,i,E)
                ‖  fork(T,i,T)†,  hungry(i)†,  ¬hungry((i+1) mod N)             → fork(E,i,T)
                ‖  fork(T,i,T)†,  hungry(i)†,  hungry((i+1) mod N)†, even(i)    → fork(E,i,W)
                ‖  fork(T,i,T)†,  hungry(i)†,  hungry((i+1) mod N)†, odd(i)     → fork(W,i,E)
                ‖  fork(T,i,E)†,  ¬hungry(i),  done((i+1) mod N)†               → fork(T,i,T)
                ‖  fork(E,i,T)†,  done(i)†,    ¬hungry((i+1) mod N)             → fork(T,i,T)
                ‖  fork(T,i,E)†,  hungry(i)†,  ¬done((i+1) mod N)               → fork(W,i,E)
                ‖  fork(E,i,T)†,  ¬done(i),    hungry((i+1) mod N)†             → fork(E,i,W)
                ‖  fork(T,i,E)†,  hungry(i)†,  done((i+1) mod N)†               → fork(E,i,T)
                ‖  fork(E,i,T)†,  done(i)†,    hungry((i+1) mod N)†             → fork(T,i,E)
                ‖  fork(W,i,E)†,               done((i+1) mod N)†               → fork(E,i,T)
                ‖  fork(E,i,W)†,  done(i)†                                      → fork(T,i,E);

        Thinking(i) ≡   user_cond , ¬done(i) → Waiting(i), hungry(i)
                    ‖   otherwise → Thinking(i);

        Waiting(i)  ≡   α , β : fork(α,i-1,Eat), fork(Eat,i,β) → Eating(i)
                    ‖   otherwise → Waiting(i);

        Eating(i)   ≡   work_to_do → work, Eating(i)
                    ‖   otherwise → Thinking(i), done(i)
        ]
    initially
        [ i : 0 ≤ i < N :: Thinking(i), Waiter(i), fork(T,i,T), Waiter(i)~Waiter(0) ]
    end
```

Figure 3: Separate synchronous transactions handling the forks

executing the barrier synchronization; and 3) each fork changed state taking only into account any message from a philosopher received by the two forks the message was sent to. To monitor the execution of the algorithm, we visualized the entire computation on a Silicon Graphics™ Personal Iris™ computer.

5. Conclusion

The development of concurrent algorithms has been profoundly affected by our perception of what computer architectures may or may not be able to implement efficiently. It is the contention of this paper that the logical simplicity of synchronous algorithms (which enjoy excellent support on current SIMD architectures) makes them attractive for many problems which are amenable to a centralized control and require both high performance and a large number of controlled asynchronous units.

6. References

1. "NCUBE Users Handbook," NCUBE Corp, Beaverton, OR (1987).

2. Aggarwal, S., Barbara, D., and Meth, K. Z., "A Software Environment for the Specification and Analysis of Problems of Coordination and Concurrency," *IEEE Transactions on Software Engineering* **14**(3), pp. 280-290 (1988).

3. Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems* (MIT Press, Cambridge, Massachusetts, 1986).

4. Banâtre, J.-P., Coutant, A., and Metayer, D. L., Eds., *The Gamma Model and its Discipline of Programming* , **15** (1990).

5. Brooks, E. D., "The Butterfly Barrier," *International Journal of Parallel Programming* **15**(4), pp. 295-307 (1986).

6. Carriero, N., and Gelernter, D., "Linda in Context," *Communications of the ACM* **32**(4), pp. 444-458 (1989).

7. Chandy, K., and Misra, J., "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems* 6(4), pp. 632-646 (1984).

8. Chandy, K. M., and Misra, J., *Parallel Program Design: A Foundation* (Addison-Wesley, New York, NY, 1988).

9. Cunningham, H. C., and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Parallel and Distributed Systems* 1(3), pp. 365-376 (1990).

10. Cunningham, H. C., and Roman, G.-C., "UNITY-style Proofs for Shared Dataspace Programs Using Dynamic Statements," Washington University, Department of Computer Science (1990).

11. Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," *Acta Informatica* 1(2), pp. 115-138 (1971).

12. Hensgen, D., Finkel, R., and Manber, U., "Two algorithms for Barrier Synchronization," *International Journal of Parallel Programming* 17(1), pp. 1-17 (1988).

13. Hillis, W. D., and Guy L. Steele, J., "Data Parallel Algorithms," *Communications of the ACM* 29(12), pp. 1170-1183 (1986).

14. IEEE, *802.3: Carrier Sense Multiple Access with Collision Detection* (IEEE, New York, 1985).

15. Lynch, N., "Upper Bounds for Static Resource Allocation in a Distributed System," *Journal of Computer and Systems Sciences* 23(2), pp. 254-278 (1981).

16. Melliar-Smith, P. M., Moser, L. E., and Agrawala, V., "Broadcast Protocols for Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems* 1(1), pp. 17-25 (1990).

17. Metcalfe, R. M., and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communication of the ACM* 19, pp. 395-404 (1976).

18. Rabin, M. O., and D., L., "On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem," *8th ACM Symposium on Principles of Programming Languages*, pp. 133-138 (1981).

19. Rem, M., "Associons: A Program Notation with Tuples Instead of Variables," *ACM Transactions on Programming Languages and Systems* 3(3), pp. 251-262 (1981).

20. Roman, G.-C., and Cunningham, H. C., "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering* **16**(12), pp. 1361-1373 (1990).