On Data Synchronization for Multiprocessors

Hong-Men Su Pen-Chung Yew

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

As the grain size becomes smaller, more parallelism can be found in most programs. However, to exploit smaller grain parallelism, more efficient synchronization primitives are needed to reduce the increased synchronization overhead. The granularity of parallelism that can be exploited on a multiprocessor system depends heavily on the type and the efficiency of the synchronization supported by the system. For medium-grain parallelism, ordered dependences such as data dependences and control dependences need to be enforced in order to guarantee the correctness of the parallel execution. Hence, data synchronization is one of the major sources of synchronization overhead in the program execution.

In this paper, we classify the synchronization schemes based on how synchronization variables are used. A new scheme, the **process-oriented scheme**, is proposed. This scheme requires a very small number of synchronization variables and can be supported very efficiently by simple hardware in the system.

Keywords: data dependences, data synchronization and parallelizing compilers.

1. Introduction

In shared-memory multiprocessor systems such as the Cray X-MP, the Alliant FX/8, the IBM 3090, the Cedar system [15] and the RP3 [21], speeding up the execution of a single job (as opposed to the improvement of system throughput) is of primary concern. In many scientific applications, getting good performance out of these systems has proven to be a non-trivial task. It requires exploiting all levels of parallelism in algorithm, program and machine architecture. It has been known for many years through Amdahl's Law that a small portion of serial code in a program can severely limit the speedup that can be obtained from parallel processing. Blindly adding more processors to a system without considering software issues will contribute very little to the system performance, and may actually be harmful due to the requirement of bulky interconnection schemes.

Exploring parallelism in algorithms and user programs has become a major thrust in parallel processing. Many compiler techniques have been developed to detect and enhance parallelism. Several successful parallelizing compilers such as VAST from Pacific Sierra, KAP from Kuck and Associates, Inc., University of Illinois' Parafrase [13], Rice University's PFC [2], IBM's PTRAN [1], to name a few, have been developed over the years. Empirical data have shown that parallelism can be exploited quite successfully in many applications on multiprocessor systems [14,16]. It has also been found that as the granularity of the parallelism becomes smaller, more parallelism can be found in most of the programs.

The trend of exploiting lower-level parallelism can be witnessed by Cray's moving away from large-grain macrotasking to support medium-grain microtasking in the Cray X-MP. However, as lower-level parallelism increases, more synchronizations become necessary. The overhead of these synchronizations determines the granularity of parallelism that can be exploited effectively. The more efficiently a system can support these synchronizations, the smaller the granularity (and hence, the more parallelism) can be exploited on the system. The data-flow computation model represents one extreme in which the granularity of parallelism is exploited down to the individual arithmetic operation level. Drastic architectural change is needed to support such systems.

In most scientific applications, loops (such as DO loops in Fortran) usually contain most of the computation in a program and are the most important source of parallelism [14]. Each loop often contains a large number of iterations with comparable running time in each iteration. The structure of these loops is very well defined, making it relatively easy to schedule them on a large number of processors.

Very often, the iterations of a loop are independent of each other and no interaction is needed when they are executed on different processes (they are called **Doall** loops in [25]). Many techniques have been developed to identify parallel loops and to transform a serial loop into a parallel loop [25]. However, even more prevalent is the case where the result produced in one iteration is used in a later iteration, or data fetched in one iteration is updated later in another itera-

This work is supported in part by National Science Foundation under Grant No.US NSF MIP-8410110, NASA NCC 2-559, the U.S. Department of Energy under Grant No. US DOE DE-FG02-85ER25001, and IBM Corporation.

tion (see Fig.2.1.a). This data access order is called data dependence [13]. Data dependence has to be enforced in order to preserve the semantics of a program. One way to enforce data dependences is to execute the loop sequentially which, of course, is very undesirable. However, if synchronization schemes are provided to allow those data dependences to be enforced among processes, all of the loop iterations can be executed concurrently (such loops are called **Doacross** loops in [8]). Of course, depending on the amount of time a processor has to wait for another processor to satisfy the data dependence, it may not be desirable to run a loop concurrently. A compiler is required to perform thorough data dependence analysis on the loop to determine which loop should be a Doacross loop. These issues have been studied quite extensively [1,4,8,25] and are beyond the scope of this paper.

Recognizing the importance of these low-level synchronizations in the exploitation of medium-grain parallelism, many recent multiprocessor systems are beginning to provide architectural support for such functions. The Cray X-MP has a set of shared semaphore registers [17], the HEP has a full/empty bit associated with each memory word [22], the Alliant FX/8 has a concurrency control bus and a set of synchronization instructions [3], and the Cedar system has a key/data scheme and a synchronization processor in each global memory module [26], etc.

In this paper, we first examine the issue of data dependences in section 2. In section 3, the existing synchronization schemes for enforcing data dependences are classified based on how synchronization variables are used. The advantages and the disadvantages of these schemes are also discussed. We then propose in section 4 a new synchronization scheme to eliminate most of the shortfalls discussed in section 3. Several examples of how to use this new scheme to exploit parallelism in programs are shown in section 5. In section 6, the hardware support needed to implement such a synchronization scheme is described. Section 7 contains our conclusions.

2. Dependences and Synchronization

2.1. Data Dependence

The dependences of programs consist of data and control dependences. Control dependence is caused by conditional branches. It can be handled by the methods similar to those for data dependences [18,26]; hence, we will only concentrate on data dependence here.

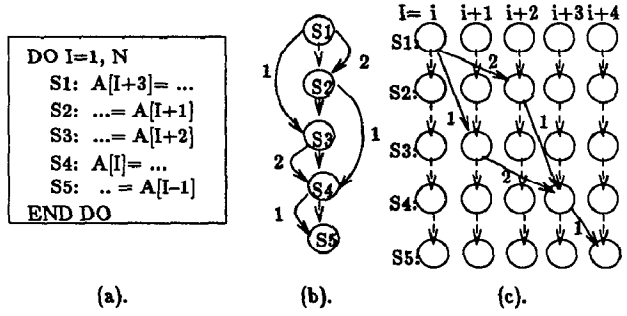
Data dependence includes (1) **flow dependence** (read-after-write), (2) **anti-dependence** (write-after-read); and (3) **output dependence** (write-after-write). These data dependences are extremely important in detecting parallelism and program restructuring. They imply a sequential order on accessing a data element, and need to be enforced in that order for correct data values. Hence, they can limit the amount of parallelism in a program.

According to their effects on program execution, the dependence relations can be categorized into two types [20]: the **ordered dependence** and the **access dependence** (or the **unordered dependence**). An **access dependence** occurs when several processes are trying to access a critical region. Each time only one process is allowed in the critical region. However, the order that those processes can enter the critical region is not restricted. It appears in many transaction-type operations and in barrier synchronization. Monitors [12], Fetch&Add [10], P's and V's operations or the like can be used effectively for this type of synchronization. Since maintaining access dependence only needs mutual exclusion, a hardware cache coherent scheme can be easily extended to enforce it [5]. Most synchronization research of the past has concentrated on enforcing this type of dependence.

In this paper, we focus on the **ordered dependence** which occurs most frequently in numerical programs, and which often determines the amount of parallelism that can be exploited. An ordered dependence exists when the order of the accesses to an object is to be enforced. A data dependence analysis can be performed on a program to obtain a **data dependence graph** using any of the schemes proposed in [2,4,25]. A data dependence graph is a directed graph. Each node in the data dependence graph is an executable statement, and each arc represents a data dependence between two nodes. The node at the tail of the arc is the *source*, and the node at the head of the arc is the *sink* of the data dependence. Since all three types of data dependences specify some kind of access order, there is no need to differentiate them when we are just trying to enforce the access order.

To simplify our discussion, we will assume that only one level of nested DO loops is to be executed in parallel. Each iteration of the loop is a **process** which can be scheduled on a processor. The idea can be extended to multiply-nested loops as well. In Fig.2.1.a, there is a loop with 4 statements: S1, S2, S3, S4 and S5. Its data dependence graph is in Fig.2.1.b. There are flow dependences S1→S2, S1→S3 and S4→S5; anti-dependences S2→S4 and S3→S4; and the output dependence S1→S4. Notice that by enforcing dependences S1→S3 and S3→S4, the dependence S1→S4 can be covered. In Fig.2.1.c, we expand the loop to show data dependences which are to be enforced between loop iterations. Each iteration is a process scheduled on a processor, and synchronization instructions are needed when these processes are accessing data elements of the array A. We call this type of synchronization **data synchronization.**

Data dependence distance is a very useful concept in data synchronization. In Fig.2.1.c, the flow dependence S1→S2 occurs in processes two iterations apart, i.e., the result stored in S1 of iteration i is used in S2 of iteration $i+2$. The data dependence distance between S1 and S2 is thus 2, and is shown next to the corresponding dependence arc in Fig.2.1.b. This data dependence distance can be easily computed by subtracting the subscript expressions of the two array references. The rest of the data dependence distances are shown in



(a). (b). (c).

Fig.2.1 (a) A loop with ordered dependences. (b) Its dependence graph. (c) Its expanded dependence graph showing only dependences around $A[i+3]$. (*Dashed lines*: execution order within iterations; *Solid lines*: cross-iteration dependences.)

Fig.2.1.b. All of the data dependences in Fig.2.1.a have a constant distance.

2.2. Data Synchronization

All data dependences must be enforced by synchronizing processes when they are accessing those shared data items. As shown in Fig.2.1, one statement can be the source or the sink of several data dependences. The following requirements must be met in order to perform correct data synchronization:

(1) The process which executes the source statement of a data dependence can signal the completion of its execution to the process which executes the sink of the data dependence only after the effect of its execution can be observed by that process. For example, in a flow dependence, the process which updates a value in its private cache must wait until the updated value is reflected in the shared memory, or reflected in a coherent cache state before it can signal the completion of its execution [9].

(2) A process executing a statement which is the sink of several data dependences (e.g. S4 in Fig.2.1.c) must wait until all of the source statements have completed before it can proceed.

Any synchronization requires some information to be transmitted between synchronizing processes. In data synchronization, this information can be stored in full/empty bits in the HEP, the synchronization registers in the Cray X-MP and in the Alliant FX/8, or the key/data pair in the shared global memory of the Cedar system. To facilitate our discussion, we use synchronization variables to represent this information. Different data synchronization schemes can incur different amount of overhead. Schemes such as HEP's full/empty bits, or Cedar's key/data pair require large amount of storage for synchronization variables in shared global memory. They are suitable for large scale multiprocessor systems. In this paper, we propose a scheme which requires smaller amount of storage for synchronization variables and is more suitable for small scale multiprocessor systems such as the Cray X-MP, the Alliant FX/8, the Encore Multimax, etc.

3. Data Synchronization Schemes

Constant-distance dependence occurs very frequently in numerical programs [25] and, due to its fixed dependence distance, can be enforced by very efficient mechanisms. Various data synchronization schemes can be characterized by the way synchronization variables are used.

3.1. Data-Oriented Schemes

In this type of schemes, at least one dedicated synchronization variable (called a **key**) is associated with each datum on which access order is to be enforced. A datum can be a scalar or an element of a structure, such as an array. A datum and its key(s) are usually stored in the same memory module, so that a data access and its related synchronization operation can be done efficiently by the memory controller. This type of scheme usually requires a large number of keys. Initializing these keys can result in significant overhead unless the number of processors in the system is large.

Based on different programming models, two types of schemes have been proposed: the **reference-based** schemes, and the **instance-based** schemes. We use the program in Fig.2.1 as an example to illustrate these schemes. Fig.3.1 shows how accessing array element A[i+3] is synchronized among processes using these schemes.

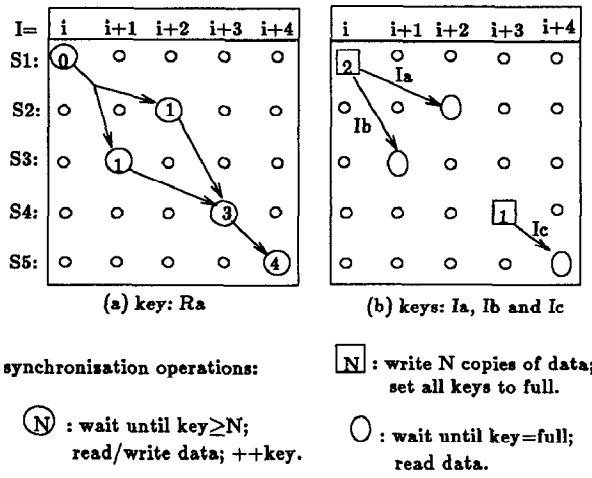


Fig.3.1 Synchronization activities of Fig.2.1 under (a) The reference-based scheme (b) The instance-based scheme.

•**Reference-Based Schemes.** Each data element is associated with a key. In Fig.3.1.a, each number in the circles indicates the **access order** which is to be compared with the key when accessing the data element. The key is initialized to 0, and after each access the key is incremented (see the instruction in Fig.3.1.a). The way the access order is checked in Fig.3.1.a allows data fetches in S2 and S3 to be performed in any order. Cedar synchronization instructions can be used to perform these operations very efficiently in a global memory module [26].

•**Instance-Based Schemes.** Each updated value of a data element is assigned a different memory location and a different key to allow read operations after the update to proceed in parallel (see Fig.3.1.b). It is similar to the single-assignment rule in the functional languages, where no output dependences and anti-dependences exist in the program. The full/empty bit in Denelcor's HEP machine [22], where each key only assumes two values (0 for empty and 1 for full), is very suitable for this scheme. **Variable renaming** is needed during compile time to remove output and anti- dependences. Multiple copies of an updated value are also needed if there are multiple reads for the updated value.

3.2. Statement-Oriented Schemes

In this class of schemes [3,18], each statement *Sa* is assigned a synchronization variable *sc* (called **statement counter** or **SC**) shared among all instances of data dependences in which *Sa* is the source (Fig.3.2.b). It enforces a sequential order such that, after process i completes its execution of *Sa*, it waits until *sc*=i-1 before it increments *sc* to i. Hence, when *sc*=i, all of the process j, j≤i, must have completed the execution of *Sa*. Initially, *sc* is set to *k-1* if the first iteration is k. For process i to execute a sink statement *Sb* (Fig.3.2.a), it has to check if each of its corresponding sources has completed, i.e., for each *Sa* with *Sa*→*Sb*, it checks if *sc*≥*i-D*, where *D* is the distance of *Sa*→*Sb*. If a statement is both a source and a sink, it must behave as a sink first. The Alliant FX/8 uses this scheme to execute Doacross loops with the support of a concurrency control bus and Advance/Await instructions [3]. Statement-oriented schemes are fairly simple to implement, but they force the updating of the SC associ-

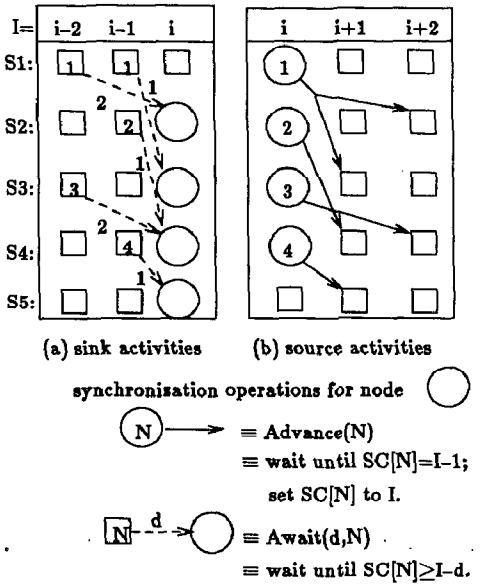


Fig.3.2 The statement-oriented scheme for Fig.2.1.

ated with each source statement to a sequential order, resulting in some loss of parallelism. For loops with many data dependences, statement-oriented schemes are also not suitable (see Example 1 in section 5).

4. A Process-Oriented Scheme

In this section, we propose a new scheme which only requires a small number of synchronization variables. There exists a **duality** between this scheme and the statement-oriented scheme. The main idea is that, instead of assigning one synchronization variable to each datum or each statement, each process (i.e., each iteration) is assigned one synchronization variable, called a **process counter (PC)**. The PC can only be updated by the process itself. A PC can be viewed as part of the state of a process which contains two pieces of information: the *process id* and the *step*. The step of a PC is updated after the completion of each source statement (see Fig.4.1). The maximum step for a PC is the total number of the source statements in each iteration. The execution of a sink statement requires it to check the PC's of all of its corresponding source statements (see Fig.4.1).

The number of iterations in a loop is usually very large. Assume we only have **X** PC's in the system. The loop needs to be **folded** to share X PC's, and each process needs to acquire a PC before it can update the PC. The process id of a PC is thus used to designate the *owner* of the PC. PC's can be arranged so that processes i, X+i, 2X+i, ..., etc. share the same process counter PC[i], where $1 \le i \le X$. Process X+i can update PC[i] only after process i releases it (by executing release_PC as explained later in Fig.4.2.a). However, a process can start its execution without obtaining a PC as long as it need not update the PC. Note that initially PC[i] should be assigned to process i, where $1 \le i \le X$.

Several useful primitives are needed in the process-oriented scheme (please refer to Fig.4.2.a): (1) **set_PC** which updates the step of the PC after the completion of a source statement (except the last one); (2) **release_PC** which gives the PC to the next process after the completion of the last

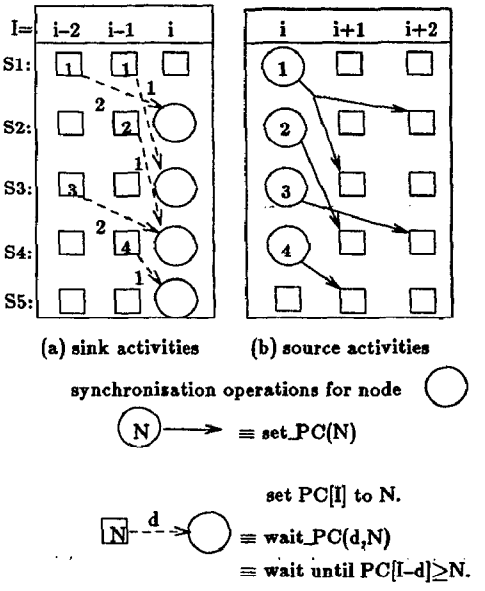


Fig.4.1 The process-oriented scheme for Fig.2.1.

```
i: the number of the iteration being executed.
X: number of process counters used.
mod: modulus operation.
PC's format: <owner, step>, and
        <w,x>≥<y,z> iff w>y, or w=y and x≥z.
Initially, PC[i]=<i,0>, 1≤i≤X.

•set_PC(current_step): /* update PC to current step. */
        PC[i mod X].step ← current_step.
•release_PC(): /* release PC for process i+X to use. */
        PC[i mod X] ← <i+X, 0>.
•wait_PC(dist, step): /* i-dist: pid of the source. */
        while( PC[(i-dist) mod X] < <i-dist, step> ).
•get_PC(): /* get the ownership of PC. */
        wait_PC(0, 0).                                  — (a) —
```

```
doacross i=1, N
  S1(i);
  get_PC(); /* wait for the PC to be available. */
  set_PC(1); /* completion of source 1. */
  wait_PC(2, 1); /* until process i-2 completes source 1.*/
  S2(i);
  set_PC(2);
  wait_PC(1, 1);
  S3(i);
  set_PC(3);
  wait_PC(1, 2);
  wait_PC(2, 3);
  S4(i);
  release_PC(); /* complete last source and release PC. */
  wait_PC(1,4);
  S5(i);
end doacross                                        — (b) —
```

Fig.4.2 (a) The primitives for the process-oriented scheme.
(b) The Doacross loop transformed from the loop in Fig.2.1.

source statment; (3) **wait_PC** which, before execution of a sink statement, spin waits until the corresponding source is completed; and (4) **get_PC** which waits for acquiring the ownership of a proper PC. Fig.4.2.b shows the code needed to syn-

myPC: the index to the owned PC.
owned: a flag denoting if a process has got its PC.
•wait_PC, get_PC, set_PC, release_PC: same as before.

```
•load_index(pid):
   myPC ← pid;                    /* use pid as index to PC. */
   owned= FALSE.                  /* Initially, PC is not owned. */
•mark_PC(current_step):
   if( not owned and PC[myPC mod X].owner < myPC )
   /* not previously owned and not transferred, don't change PC. */
        return;
   set_PC(current_step);
   owned= TRUE.
•transfer_PC():
   if( not owned )  get_PC();
   release_PC().                  /* give ownership to next process. */
```

Fig.4.3 The improved primitives for the process–oriented scheme.

chronize the loop in Fig.2.1. The transfer of the ownership is accomplished by process i executing **release_PC** on the completion of its last source statement, and by process X+i executing **get_PC** before it first updates the PC.

Notice that in the statement–oriented scheme, the ownership of a SC is shared in turn by all instances of the same source statement. Since different instances of the same source statement very often can be executed simultaneously as in Fig.2.1, such "horizontal" sharing introduces unnecessary delay due to waiting for ownership. In other words, the process i must wait for process i–1 to release the ownership of each SC, implying that process i must wait for the completion of all the processes before it. If for some reason one process delays its release of the SC (e.g. executing a longer branch), all later processes will be affected. In the process–oriented scheme, a PC is shared by all source statements in the same process. Since statements are assumed to execute sequentially within a process, this "vertical" sharing of a PC can never result in such delay. In the folded version, the delay due to waiting for ownership of a PC currently belonging to process i can happen only in processes X+i, 2X+i, ..., etc. However, this occurs less frequently than in the statement–oriented scheme if X is large enough.

Actually, if we study it more carefully, the requirement of acquiring a PC before executing the first source statement as in Fig.4.2.b can be further relaxed. We can have an improved scheme based on the new primitives in Fig.4.3. On completion of each source statement (except the last one), a process tests the ownership of the PC. If it owns the PC, then a new state is marked; otherwise, it proceeds without waiting for the availability of the PC. This is done by executing **mark_PC**. However, to signal the completion of all its source statements and to transfer the ownership of the PC to the next process, it must execute **transfer_PC** after the completion of the last source statement. Executing **transfer_PC** guarantees that a process has owned a PC and has the right to transfer the PC to the next owner. In Fig.4.3, another primitive **load_index** is also introduced. It saves the index of a PC in an internal variable *myPC* to be used by others primitives and resets the flag *owned*. In the improved scheme, the new primitives **mark_PC** and **transfer_PC** replace the **set_PC** and the **release_PC**, respectively; while the **load_index** can substitute the **get_PC**. (In fact, load_index can be the first statement of the loop body.)

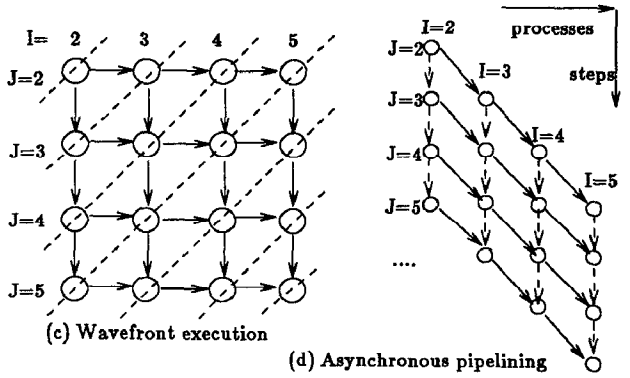
5. Applications

Because the process–oriented scheme uses only one synchronization variable per iteration for all of its dependences, it can handle much more complicated cross–iteration dependences, especially when a Doacross loop contains other serial loops or procedure calls, and when loop boundaries need to be considered.

•**Example 1.** a Doacross loop enclosing a serial loop:

```
DO I=2, N                          doacross i=2, N
                                     load_index(i);
   DO J=2, N                         do k=2, N, G
                                       wait_PC(1,k);
                                       do j=k, k+G-1
   S1: A[I,J]=A[I-1,J]                   S1(i,j);
             +A[I,J-1]                 end do
                                       mark_PC(k);
   END DO                            end do
                                     transfer_PC();
END DO                             end do
```

(a) The original loop. (b) The Doacross loop.



(c) Wavefront execution (d) Asynchronous pipelining

Fig.5.1 Synchronizing a Doacross loop enclosing a serial loop.

Fig.5.1.a is a simplified four–point relaxation code. Fig.5.1.c depicts the well known **wavefront** method which requires loop index transformation. A barrier synchronization is needed between two consecutive wavefronts. However, the execution of a barrier requires that processors be busy–waiting at the barrier until all of the processors arrive. An alternative is to use an **asynchronous pipelined** method as shown in Fig.5.1.d, in which we serialize the inner loop as a process and execute the outer loop as a Doacross loop. The two methods will have the same number of parallel steps; however, the efficiency and the processor utilization is much better in the asynchronous pipelined method.

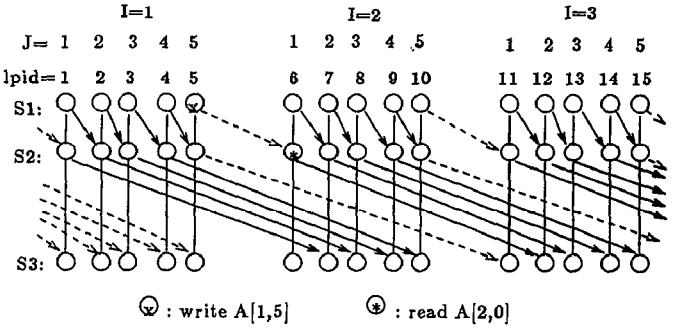
Since there are N–1 synchronization points between two consecutive processes (Fig.5.1.d), this implies that N–1 SC's are needed to get the maximum parallelism if we use the statement–oriented scheme. However, in practice, N is very large which makes the statement–oriented scheme perform poorly when the number of SC's is limited. By using the process–oriented scheme, we are able not only to synchronize the loop with a small number of PC's, but also to exploit the parallelism very effectively. We can also reduce the amount of synchronization needed between successive iterations of I by grouping G iterations in the J loop as shown in Fig.5.1.c. (Assume (N–1)/G is an integer.) Some extra delay will be incurred between the successive iterations of the I loop; however, the amount of synchronization can be reduced

significantly due to the increase of granularity.

•**Example 2.** A multiply-nested Doacross loop:

```
DO I=1, N                      doacross i=1, N
 DO J=1, M                       doacross j=1, M
                                   load_index((i-1)*M+j);
  S1: A[I,J]= ...                  s1(i,j);
                                   mark_PC(1);
  S2: B[I,J]= A[I,J-1] ...         wait_PC(1,1);
                                   s2(i,j);
                                   transfer_PC();
  S3: ... = B[I-1,J-1]             wait_PC(M+1,2);
                                   s3(i,j);
 END DO                          end do
END DO                         end do
```

(a) The original loop (b) The Doacross loop



(c) The expanded dependence graph with M=5.

Fig.5.2 Synchronizing a multiply-nested Doacross loop.

One of the major problems in data-oriented schemes is that they are quite awkward in handling loop boundaries when there are multiple loop nestings. As shown in Fig.5.2.c, at loop boundaries (when J=1), A[I,J-1] in S2 and B[I-1,J-2] in S3 do not depend on any source as shown by dashed lines. Whereas, in the rest of the iterations, A[I,J-1] in S2 depends on A[I,J] in S1, and B[I-1,J-2] in S3 depends on B[I,J] in S2 as shown by solid lines. These boundaries need to be handled explicitly by a lot of extra code and overhead.

Using the process-oriented scheme, these loop nestings can be implicitly coalesced to obtain a **linearized process id** (*lpid*) as the index to the PC. When loop index set is equal to (i,j), the *lpid* is *(i-1)*M+j*. After that, the loop can be executed as a singly-nested loop without worrying about loop boundaries (Fig.5.2.b). However, implicit coalescing can introduce extra dependences (shown as dashed lines in Fig.5.2). Some parallelism may be lost from these extra dependences, but the complexity of detecting boundaries is avoided. That overhead can be $O(r^2d)$ per iteration, where r is the number of occurrences of an array variable and d is the depth of the nested loop [24].

Note that data-oriented schemes still have the boundary problem even after the loop is linearized. This is because, in those schemes, synchronizations are done on each data element. The number of times each data element is accessed (or synchronized) in a loop is fixed, and may be different for those data elements referenced at the loop boundaries. Linearization cannot change the number of times a data element is accessed. Furthermore, introducing extra accesses for those data elements at the boundaries to make the number of synchronizations the same for all data elements requires the testing of boundaries anyway. It is not as easy as in our process-oriented scheme.

•**Example 3.** Dependence sources in branches:

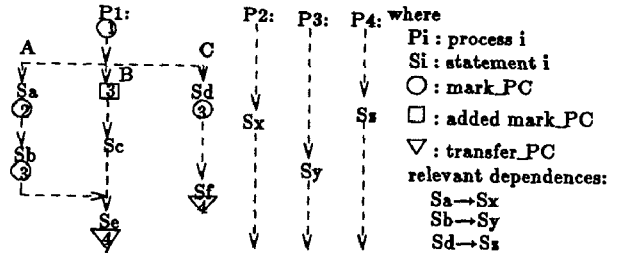
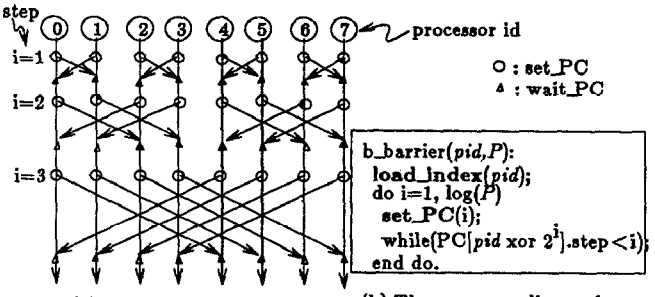


Fig.5.3 Synchronizing dependences with sources in branches.

When there are conditional branches, some of the data dependences may not exist because a branch may not be taken. One solution is as follows: if a synchronization primitive changes a synchronization variable in one path, the synchronization variable must also be changed in all other paths to allow the effect to be the same no matter which branch was taken.

In Fig.5.3, P1 executes **transfer_PC** before it completes its execution. Every sink of P1 eventually can proceed. However, P1 should inform the sinks to proceed as soon as possible. So in Fig.5.3, after Sd in branch C, **mark_PC**(3) is executed instead of **mark_PC**(2); and **mark_PC**(3), though not required, is added as the first statement in branch B.

•**Example 4.** Implementing a butterfly barrier:



(a) The computation graph (b) The corresponding code

Fig.5.4 Implementing a butterfly barrier.

A butterfly barrier (Fig.5.4.a), which can remove the hot-spot effect, performs better than a counter-based barrier even in a small bus-based system [6], and it needs no atomic operation. Using the process-oriented scheme, it requires fewer synchronization variables and operations than those needed in [6]. Procedure b_barrier() in Fig.5.4.b is called by each processor with different *pid*, where P, the number of processors, is a power of 2 and xor is a bitwise exclusive-or operation. Since each process corresponds to a processor in this case, no folding is needed. Thus the computation involved in obtaining the ownership can be eliminated. The *while* statement in Fig.5.4.b is a modified version of **wait_PC**. Note that with a minor modification, b_barrier() can work even when P is not a power of 2 [11].

•**Example 5.** Executing phases of computation with local communication:

```
fft(pid, P):        /* pid, P: same as in b_barrier(). */
  load_index(pid);
  do i=1, log(P)
     BASIC_FFT(pid, i, P);
     set_PC(i);
     while( PC[pid xor 2^i].step < i );
  end do.
```

The example shown here is an FFT. If we partition the data used in the FFT into chunks equal to the number of processors, the computation pattern is the same as a butterfly barrier in Example 4, except that an additional BASIC_FFT() is performed in each stage. However, since communication only takes place between two processors in each stage, there is no need for a global barrier as in [7]. Instead, in each stage, after each processor completes its computation in BASIC_FFT(), it only waits for another processor with which it exchanges data. Procedure fft() is intended to be called in the same way as b_barrier() in Example 4. In fact, the process-oriented scheme is very suitable for code with many phases of computation. After each phase, a limited amount of communication is needed in each sub-group. Another example is the discretization method for solving partial differential equations [19], in which a process only needs to synchronize with processes computing its neighboring regions.

Several comments can be made about the process-oriented synchronization scheme. First, it can be incorporated into a concurrentizing compiler using algorithms similar to [18]. Second, it can also be used as a new paradigm in parallel programming. Only one synchronization variable is needed in each process to handle all ordered dependences. Third, it fits very well in dynamic scheduling schemes such as processor self-scheduling [24], where better load balancing among processors can be achieved. As a matter of fact, dynamic scheduling is assumed in all of the examples shown above.

6. Hardware Considerations

Notice that because we are trying to support medium-grain parallelism, it is not efficient to use context switching whenever a synchronization operation needs to wait. **Busy-waiting** is more appropriate in this case. A good scheduling policy such as proposed in [23] can reduce such busy-waiting even further.

Excluding busy-waiting, each access to the source or the sink of a data dependence requires only one extra access to its corresponding PC. The extra traffic incurred is small, but the parallelism obtained can be very significant. The PC's could be incorporated in a hardware-maintained coherent cache system, even though they may be purged out of a cache.

To reduce the access time of a PC and the impact of busy-waiting traffic, we can use a dedicated synchronization bus and some synchronization registers to store the PC's as in Alliant FX/8. Each processor keeps an image of all PC's in its local synchronization registers. Whenever a PC is updated in one processor, the new value is broadcast via the bus to all of the other processors so that the local image of the PC in each processor can be updated. Notice that, since a PC needs to be updated only after the source statement is completed, the amount of such traffic is no worse than that in the main data bus. Busy-waiting operations can then be performed on the local copies of the PC's without introducing extra traffic to the synchronization bus. Also, since the process id is used quite frequently in our synchronization primitives, a special register **myPC** for each processor can be used to store its value. A local status bit **owned** is also helpful to indicate if it has already owned its corresponding PC. **Owned** is set when a process updates its PC in mark_PC, and is reset in load_index. The proposed scheme works best if the number of PC's (i.e., X) equals to a power of 2 and is a small multiple of the number of processors. The modulus operation needed in computing the index of a PC can then be done easily by taking the lower bits of a process id.

It is worth noting that the primitives need **not** be **atomic**, because each PC is monotonically incremented by only one processor at any time and wait_PC waits for the PC to exceed (not stay at) a certain value. This makes the primitives much easier to implement. Also, the two fields in a PC, i.e., **owner** and **step**, need not be updated simultaneously (this can reduce the bus width). There are two reasons for this. First, in our scheme, it is impossible for two processes to update the same PC at the same time. Second, the read of a local PC (by wait_PC) and the update of the PC (by a mark_PC or transfer_PC) can proceed in any order without changing the desired synchronization behavior. Assume the value of the PC is <i,j1>. An update will change it to either <i,j2> where j2>j1 (by a mark_PC), or <i+X,0> (by transfer_PC). In the former case, a read either gets <i,j1> or <i,j2>. If retried, it will eventually get the value <i,j2>. In the latter case, if **step** is updated first, then the transition of the PC's values is <i,j1>→<i,0>→<i+X,0> which again will not cause any undesired side effect.

Further reduction in the broadcast transactions (i.e., writes) is possible because a PC is updated by the sequence of mark_PC(1), mark_PC(2), ..., and transfer_PC from the same process. Each later write covers all previous ones. However, individual information should be provided as soon as possible. As seen in section 4, a mark_PC need not update the PC if the ownership has not been obtained. However, its ownership is guaranteed by the final transfer_PC in each process. This can reduce the number of global synchronization operations and unnecessary waiting. A similar improvement can also be applied to hardware. For example, an issued write need not be sent out if a second write to the same PC arrives before the former has gained the bus access, thus avoid the extra bus traffic.

The above proposed implementation is similar to the concurrency control bus in the Alliant FX/8. However, by allowing a synchronization variable (PC) to be indexed by a variable[1] (i.e., **pid** needed in the primitives is stored in register **myPC**), it significantly improves the functionality of our proposed primitives. Hardware and software reduction of synchronization and waiting operations also become easier in our scheme.

The advantages of the process-oriented scheme can be summarized as follows: (1) It eliminates the use of barrier synchronization to implicitly enforce data dependences when the latter is not suitable (Example 5). Memory contentions (i.e., the hot-spot effect) and the inefficiency caused by waiting for the last processor to complete in a barrier synchronization can be avoided; (2) Compared to the statement-oriented scheme, it eliminates unnecessary serialization of consecutive iterations

[1] The index to a synchronization register accessed by Alliant's Advance and Await must be a constant.

and can handle loops with many dependences more efficiently (Example 1); and (3) It can handle multiply-nested loops without the overhead of checking loop boundaries as needed in data-oriented schemes (Example 2). It also reduces the number of synchronization variables and the overhead associated with initializing these variables very substantially.

7. Conclusions

In exploring low-level parallelism, efficient synchronization schemes are needed. Ordered dependences are very important in exploring medium-grain parallelism. They require synchronization mechanisms which are very different from the transaction-type of synchronization such as maintaining the exclusive usage of a critical region. To enforce ordered dependences, efficient data synchronization is essential.

We have identified issues which are important in data synchronization. By the method of exchanging synchronization information, several synchronization schemes are categorized and compared. A new process-oriented scheme which requires only a small number of synchronization variables is proposed. It can be supported by very simple hardware in the system. Several applications of this scheme are presented to show its advantages over the previously proposed schemes.

References:

[1]. F. Allen, M. Burke, P. Charles, R. Cytron and J. Ferrante. *An Overview of the PTRAN Analysis System for Multiprocessing.* **Int. Conf. on Supercomputing** (June 1987).

[2]. R. Allen and K. Kennedy. "PFC: A Program to Convert Fortran to Parallel Forms", Rep. MASC-TR82-6, Rice Univ., 1982.

[3]. Alliant. **FX/Series Architecture Manual**. Alliant Computer Systems Corp., 1986.

[4]. U. Banerjee. "Speedup of Ordinary Programs", Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-989, 1979.

[5]. Bitar and A. Despain. *Multiprocessor Cache Synchronization: Issues, Innovations, Evolution.* **Int. Symp. on Computer Architecture** (June 1986) pp. 424-433.

[6]. E.D.Brooks III. *The Butterfly Barrier.* **Int J. of Parallel Programming** (1986) vol. 15-4, pp. 295-307.

[7]. Z.Cvetanovic. *Performance Analysis of the FFT Algorithm on a Shared-Memory Parallel Architecture.* **IBM J. Res. Develop.** (July 1987) pp. 435-451.

[8]. Ron Cytron. *Doacross: Beyond Vectorization for Multiprocessors.* **1986 Int. Conf. on Parallel Processing** (Aug. 1986) pp. 836-845.

[9]. M. Dubois, C. Scheurich and F. Briggs. *Synchronization, Coherence, and Event Ordering in Multiprocessors.* **Computer** (Feb. 1988) pp. 9-21.

[10]. A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph and M. Snir. *The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer.* **IEEE Trans. Comput.** (Feb. 1983) pp. 175-189.

[11]. D. Hensgen, R. Finkel and U. Manber. *Two Algorithms for Barrier Synchronization.* **Int. J. of Parallel Programming** (1988) vol. 17-1, pp. 1-17.

[12]. C.A.R Hoare. *Monitors: An Operating System Structuring Concept.* **CACM** (Oct. 1974) pp. 549-557.

[13]. D. Kuck, R. Kuhn, D. Padua, B. Leasure and M. Wolfe. *Dependence Graphs and Compiler Optimizations.* **ACM Symp. on Principles of Programming Languages** (July 1981).

[14]. D. Kuck, A. Sameh, R. Cytron, A. Veidenbaum, C. Polychronopoulos, G. Lee, T. McDaniel, B. Leasure, C. Beckman, J. Davies and C. Kruskal. *The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance.* **1984 Int. Conf. on Parallel Processing** (Aug. 1984).

[15]. D. Kuck, E. Davidson, D. Lawrie and A. Sameh. *Parallel Supercomputing Today and the Cedar Approach.* **Science** (Feb. 1986) vol. 231, pp. 967-974.

[16]. M. Kumar. *Effect of Storage Allocation/Reclamation Methods on Parallelism and Storage Requirements.* **Int. Symp. on Computer Architecture** (June 1987) pp. 197-205.

[17]. J. Larson. *Multitasking on the Cray X-MP-2 Multiprocessor.* **Computer** (July 1984) pp. 62-69.

[18]. Samuel Midkiff and David Padua. *Compiler Algorithms for Synchronization.* **IEEE Trans. Comput.** (Dec. 1987) pp. 1485-1495.

[19]. D.M.Nicol and F.H.Willard. *Problem Size, Parallel Architecture, and Optimal Speedup.* **Int Conf. Parallel Processing** (Aug. 1987) pp. 347-354.

[20]. Anita Osterhaug. **Guide to Parallel Programming on Sequent Computer Systems.** Sequent Computer Systems, Inc., 1986.

[21]. G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton and J. Weiss. *The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture.* **1985 Int. Conf. on Parallel Processing** (Aug. 1985) pp. 764-771.

[22]. B. J. Smith. *A Pipelined, Shared resource MIMD Computer.* **1978 Int. Conf. on Parallel Processing** (Aug. 1978) pp. 6-8.

[23]. P. Tang, P. C. Yew and C. Q. Zhu. *Impact of Self-Scheduling Order on Performance of Multiprocessor Systems.* **1988 Int. Conf. on Supercomputing** (July 1988) pp. 593-603.

[24]. Peiyi Tang. "Self-Scheduling, Data Synchronization and Program Transformations for Multiprocessor Supercomputers", Ph.D. Thesis, CSRD Report #809, Univ. of Illinois, Urbana-Champaign., 1988.

[25]. M. Wolfe. "Optimizing Supercompiler for Supercomputers", Ph.D. thesis, Rep.82-1105, DCS, Univ. of Illinois at Urbana-Champaign, 1982.

[26]. Chuan Qi Zhu and Pen-Chung Yew. *A Scheme to Enforce Data Dependence on Large Multiprocessor Systems.* **IEEE Trans. Software Eng.** (June 1987) pp. 726-739.