Events that are reissued:

- Key algorithm design issue is coordinating clears of posted can proceed immediately.
- Any task that starts waiting after the event has been event may proceed.
- When an event is posted ALL tasks waiting for the tasks that execute a wait for a particular event spin off posted a second time.
- Once an event is posted it must be cleared before being Events can be posted, cleared, and waited for.

Synchronization:

Advance/Await is a loop-based form of Event-based
wait for an event is a key question.

Knowing or determining how many tasks will eventually

wait for the event have received the post.

Clearing must occur after all tasks who will eventually

Clearing a cleared event is OK for initialization.
assumes the task posting the event is still active.

This has proceeded and the event can be cleared. (This
let the task that posted the event know that all waiters
A feedback event from all waiting tasks can be used to
the one to clear.

use a lock-protected counter to determine which task is
If multiple tasks of known number are waiting, you can
post the event that it has been cleared.

dependent string of events to tell the next task who is to
responsible for clearing it. (Note there still must be some
If only one task is waiting for an event then it is often


diners to eat current course. Assume there are 3 dimers.

processed-with-course by which the host informs the
diner is required. Along with an event
before eating it. An event course-arrived-i for each
diners so that all wait until everyone has the next course
Suppose a host at a diner wants to coordinate the

A SIMPLE EVENT RESUSING EXAMPLE
end do
end course

POST(Proceed)
has arrived for host
when next course
clear(course-arrived-1)

wait(course-arrived-3)

wait(course-arrived-2)

wait(course-arrived-1)
clear(Proceed)
do until courses finished
DINNER-II

HOST
Post as two.

(protagonist viewing one proceed on the previous one. Eroniously view the next course while others are still dinner would start on the next course before it has been cleared. The one and wait for proceed before it has been cleared. The next course.

- A last diner may finish the previous course, get the next received it.
- A last host may clean the proceed before all diners have
- Iteration does not work. Simply clear at the bottom or top of the host.
- There is a problem with the clearing of the proceed and the first proceed position correctly but note that
- This code handles the first set of course-arrived events.
Use two proceed events – even and odd courses.

Proceed with the next course by a fast diner.

Make sure all diners see it and having it interpreted as

The problem is the proceed event remaining posted to

ahead.

The course does not help, one dinner could still face

Simply add the feedback event for each dinner finishing

•
end do
i=i+1

dine

end course
clear(course-arrived-t)
end if

wait(procc-odd)
else
wait(procc-even)

if (i is even)
post(course-arrived-t)
has arrived for dinner-
next course

when next course

do until courses finished

i=1
DINNER-I

end do

host
started and who has finished a course.

A simpler strategy would use locked counters for who has proce-even or odd for the previous course can be cleared.

are all finished with the previous course and the next course has arrived for all diners, therefore they next course it has already received the events indicating when the host is ready to post proce-even or odd for the proce-even or odd for the next course.

Diners who race ahead are stopped by the cleared
their course-arrived events have been cleared. The subsequent proceed event tells all of the diners that cleared by the host since he is waiting for it. The problem is that the course-arrived events should be cleared.

course gets to anyone other than the host. course proceed will be issued by the host before any new diners have cleared the arrived events. Therefore a new clear proceed and get back to the wait. before the last host (say one who does not eat the courses) may there is still a problem with the course events.
end of the loop.

processor proceeds to the barrier synchronization at the

is nonpositive then the work is exhausted and the

loop iteration with that value as the index. If the counter

value the counter reads is positive then it executes the

reads the counter and then decrements it by 1. If the

fetch-and-decrement(counter,1) which indivisibly

When a processor needs work it executes a

to the total loop count N.

Dynamic scheduling of a loop uses a counter initialized

hardware support — fetch-and-op.

more general synchronization primitives that often have

requiring atomic updates to integer counters can use

Dynamic scheduling, barriers and any other situation
need work.

beginning of the loop or whenever several processors can be trying to access the counter — especially at the

This causes a hot spot in memory since many processors.
usual ALU integer/logical instructions.
do is usually add or decrement, but proposals and
processors to both create and consume work.
a more general scheduling algorithm is used that allows
schedule operation. The latter is especially important in
hierarchy of locks to complete for before getting to
processor when they are brought into the parallel loop;
Software solution – scheduler initializes work for each
Hardware solution – combining networks
COMBINING FETCH AND ADD'S
actually restarts parallelism.

In practice some extra logic is required to see everyone
the barrier.
sequentially as in a join or it can set $P \rightarrow C$ to complete
the master (all others must be spinning). It can proceed
If the value fetched is $1$ then the processor continues as
parallel loop cooperation is indiciated again with $C = P$.

If the value fetched is positive the processor spins until

**fetch-and-decrement($C, I$).**

After finishing all work a processor performs
begins.

Initialize $C \rightarrow P$ for $P \rightarrow C$ when parallel loop

Barsiers can be handled as well.
Synchronization is required to ensure correct summation.

on its list.

contributions from all devices or elements that have (i, j)

The value of A in position (i, j) (i, j) must sum the

\[ \{ (i, j, 1), (i, j, 2), \ldots, (i, j, n) \} = T \]

which it contributes:

element, e = 1, 2, \ldots, M, therefore has a list of positions to contribute to values in multiple positions in A. Each

each

\[ \sum_{i, j, k} A(i, j, k) = \text{value} \]

Assume that there are M elements or devices that each

values must be computed.

Assume a matrix A(1:N, 1:N) must be assembled, i.e., its

A Typical Application Code Example
The solution requires one barrier and extra space.

- can be used for each element to minimize extra storage.
- contributing to each position then variable length lists
- If there is a wide range of the number of elements
  \[ \{\ell, \ell'\} \]
  sum the, at most, \( S \) values to get the final value of each
  position.

After a barrier, a parallel loop over the \( N \) positions can
  and write to these locations at the same time.

and write to these locations at the same time.

for its contributions to each position and may compute

\( A(1:N, 1:N, S) \).

Each element then has a unique address

memory for each position \( \{\ell, \ell'\} \), i.e., dimension

elements each position sums, \( S \), and create \( S \) spaces.

Easiest solution: determine maximum number of
Problems: arithmetic delays when two positions that are not actually distant.

• i.e., a global critical section.

Only one processor will update the matrix at any time.

• Before proceeding,

matrix requires the processor to seize a lock location when a contribution is to be made to any element in the.

• When a contribution is to be made to any element in the.

their associated positions.

decreases and have each compute their contributions to

the simplest approach is to start a parallel loop over the M.

• Rather than use extra space, synchronization points can

be used.
Lock-protected regions:

- The positions updated over the elements and the distribution may reduce delay time depending on the distribution.
- (element are needed)
- It adds synchronization points (multiple locks per element)
- This requires extra space for the extra locks.
- used

Solution: Add extra locks, e.g., a lock per row or column.
- Write on empty leave full.
- Read on full leave empty.
- Read on full leave full.
- Normal; i.e., the bit is ignored.

Reads and writes have several modes:

FULL/EMPTY BIT.

Each storage location has associated with it a
per element quite natural.

both have special hardware support that makes a lock

The old HEP machine and the very new TERA machine
Update

- Easy to protect individual elements for indivisible
- Ideal for producer/consumer pairs.
- Actions indivisibly check bit, read/write, update bit.
initial value of A is available.

Assume that the full/empty bit is set to full when the
represented \texttt{READ-FE}(T).

Let \texttt{READ} on full leave empty of a location L be
represented \texttt{WRITE-EE}(T,C).

Let \texttt{WRITE} on empty leave full of value C to location L be

single shared A.

\textit{i.e., each processor might have a local to add to a}
\texttt{D} to have added to it indivisibly,

Suppose location A is to have added to it indivisibly.
back to A.

then performs the addition and writes the updated value

• The processor that succeeded in setting the bit empty

the bit is reset to full.

• to stop all others from reading, i.e., they will spin until

READ-FE(A). One will succeed and set the bit to empty

• All processors competing for A attempt to do the

indivisible update.

WRITE-EF(A, READ-FE(A)+D) accomplishes the
Requires special hardware support.

Essentially a memory-based spin-wait.

If none are pending then whenever the next one is issued

the location

the bit to full and the pending READ-PE complete for

When the update is complete then the WRITE-PE sets

processor fetched the initial value of A.

on empty and the bit was set to empty when the

This must succeed since the WRITE-PE will only write
simultaneously without synchronization problems. Therefore their contributions can be computed and added elements that update independent sets of positions and Nodes that are not connected represent devices or positions in \( A \) of course.

(They may both update several position in the matrix \( A \).) An edge exists between two nodes if and only if the two elements corresponding to the nodes update the same

An edge exists between two nodes if and only if the two

A graph is formed with a node for each element or device.

\[ \text{Preprocessing.} \]

\[ \text{A compromise approach that uses multiple barriers and no critical sections or full/empty bits requires some} \]

\[ \text{A compromise approach that uses multiple barriers and} \]

\[ \text{A compromise approach that uses multiple barriers and} \]
A barrier is assumed between each parallel loop.

- nodes of the $i$-th color.

- series of $i$ parallel loops the $i$-th of which processes all

If there are $k$ colors the matrix $A$ can be computed by a

- by an edge.

Nodes with the same color are therefore not connected

- and sequential methods to do this.

- an edge have different colors. (There are various parallel

Color the graph so that two nodes that are connected by

-
reused for others.
the storage used for the intermediate results can then be
that the effects of the intermediate results are felt and
synchronization operations can be used to make sure

Alternatively, extra (and typically more complicated)

• Increase parallelism

future time tends to simplify synchronization and
Adding extra space to store them until a convenient

• Intermediate results

Essentially we are asking, "What do we do with
general activity in designing parallel algorithms
The previous discussions are specific examples of a
preferences in terms of synchronization primitives.
portable code across machines that have distinct
compromises in performance are often needed to have
reasonable performance.
the architecture in an efficient manner in order to achieve
constructs provided by the programming language and
The form of this synchronization must be mapped to the