Barrier Synchronization

Consider co-in-while structure

```
while( true ) {
        co [i = 0 to n − 1] {
                code to implement task i } }
```

To convert to while-in-co form, we must synchronize:

```
co [i = 0 to n − 1] {
        while (true) {
                code to implement task i
                wait for all n tasks to complete }
}
```

Mutual inclusion

---

init: **int** $s[n] := [(n)0]$; # Number of rounds started.
init: **int** $c[n] := [(n)0]$; # Number of rounds completed.

P_i :	P_j :
$s[i] := 1$;	$s[j] := 1$;
Round(i,0)	Round(j,0)
$c[i] := 1$;	$c[j] := 1$;
Barrier	Barrier
$s[i] := 2$;	$s[j] := 2$;
Round(i,1)	Round(j,1)
$c[i] := 2$;	$c[j] := 2$;
Barrier	Barrier
...	...

Working: $s[i] = c[i] + 1$

In barrier: $s[i] = c[i]$

Thus: $c[i] \leq s[i] \leq c[i] + 1$, for all i, is a global invariant.

Desired Global Invariant: We require the barrier to delay the start of round $k+1$ until all threads have completed round k.

$$s[i] = k+1 \Rightarrow c[j] \geq k, \text{ for all } i, j, k$$

Equivalently:

$$s[i] \leq c[j] + 1, \text{ for all } i, j$$

Define[1]

$$\min(c) \triangleq \min_{j \in \{0,..n\}} c[j]$$

The desired global invariant can be stated as:

$$s[i] \leq \min(c) + 1, \text{ for all } i$$

An Abstract solution

---

int $s[n] := [(n)0], c[n] := [(n)0]$;
process Worker[$i = 0$ **to** $n-1$] {
    while (true) {
        ## $s[i] = c[i] = \min(c)$
        $s[i] \mathrel{+}= 1$;
        code to implement task i
        $c[i] \mathrel{+}= 1$;
        $\langle \mathbf{await}(\min(c) = c[i]) \rangle$ }
}

[1] $\{0, ..n\}$ means the first n natural numbers.

Proof sketch:

- Local correctness.
  - We need to show that $s[i] = c[i]$ is a loop invariant.
  - And also that $c[i] = \min(c)$ is a loop invariant.
- Noninterference.
  - No other thread changes $s[i]$ or $c[i]$, and so $s[i] = c[i]$ is not interfered with.
  - Let $j \neq i$ We must check that
    $$\vdash \{c[i] = \min(c)\}\ \ c[j] \mathrel{+}= 1\ \ \{c[i] = \min(c)\}$$
- Global invariance
  - '$s[i] \leq \min(c) + 1$, for all i' is true initially since $0 \leq 1$
  - We must check '$s[i] \leq \min(c) + 1$, for all i' is preserved by each assignment. (a) For all $i \in \{0, ..n\}$
    $$\begin{array}{ll} \vdash & \{s[i] \leq \min(c) + 1 \wedge s[i] = c[i] = \min(c)\} \\ & s[i] \mathrel{+}= 1 \\ & \{s[i] \leq \min(c) + 1\} \end{array}$$
    $\Leftarrow$ Subset the precondition
    $$\vdash \{s[i] = \min(c)\}\ \ s[i] \mathrel{+}= 1\ \ \{s[i] \leq \min(c) + 1\}$$
    and (b) for all $i, j \in \{0, ..n\}$
    $$\vdash \{s[i] \leq \min(c) + 1\}\ \ c[j] \mathrel{+}= 1\ \ \{s[i] \leq \min(c) + 1\}$$

Lots of Counters

Data refine the abstract solution

- Augment with an infinite array *counts*. $counts[k]$ is the number of threads that have completed round k. Formally we have a global invariant

  $$counts[k] = |\{j \mid c[j] \geq k\}| \text{, for all } k > 0$$

  Introduce local variable k to track $s[i]$.
- Transform: Replace '$\min(c) = c[i]$' by '$counts[k] = n$'
- Demote c and s to ghost variables.

---

int $s[n] := [(n)0], c[n] := [(n)0]$;
int $counts[1..\infty] := ([1..\infty]0)$
process Worker[$i = 0$ **to** $n - 1$] {
    for[$k = 1$ **to** ∞] {
        $s[i]$ += 1 ;
        code to implement task i
        $\langle counts[k]$ += 1; $c[i]$ += 1; $\rangle$
        $\langle$**await**$(counts[k] = n)\rangle$
        ## $c[i] = \min(c)$ }
}

Proof that the transformation of the await condition is correct:

- Recall that $counts$ and c are linked by the abstraction relation

$$counts[k] = |\{j \mid c[j] \geq k\}|, \text{ for all } k$$

- Now calculate:

$$\begin{aligned}
& counts[k] = n \\
= \;& \text{The abstraction relation} \\
& |\{j \mid c[j] \geq k\}| = n \\
= \;& \text{Since the size of the } c \text{ array is } n \\
& c[j] \geq k, \text{ for all } j \in \{0, ..n\} \\
= \;& \text{Since } k \text{ is equal to } c[i] \text{ at this point} \\
& c[j] \geq c[i], \text{ for all } j \in \{0, ..n\} \\
= \;& \text{Definition of } \min(c) \\
& c[i] = \min(c)
\end{aligned}$$

Three Shared Counters

---

int $count0 := 0, count1 := n, count2 := n$;
process Worker[$i = 0$ **to** $n - 1$] {
    while(*true*)
        code to implement task i
        $< count0 \mathrel{+}= 1 ; >$
        $< count1 \mathrel{-}= 1 ; >$
        $<$ **await** $(count0 = n)$; $>$

        code to implement task i
        $< count1 \mathrel{+}= 1 ; >$
        $< count2 \mathrel{-}= 1 ; >$
        $<$ **await** $(count1 = n)$; $>$

        code to implement task i
        $< count2 \mathrel{+}= 1 ; >$
        $< count0 \mathrel{-}= 1 ; >$
        $<$ **await** $(count2 = n)$; $>$ }
}

This still requires atomic increment and decrement operations.

One counter

Since only counts[k] is referenced, (and all k's are kept almost in sync), we only need one count variable. A coordinator resets it to 0 so it can represent the next item of counts.

```
# int s[n] := [(n)0], c[n] := [(n)0] ;
# int counts[1..∞] := ([1..∞]0) ;
int count := 0 ;
process Worker[i = 0 to n − 1] {
    for[ k = 1 to ∞] {
        ## count = counts[k]
        s[i] += 1 ;
        code to implement task i
        < count += 1; counts[k] += 1; c[i] += 1; >
        < await (count = 0 ) >
        ## c[i] = min(c) and count = counts[k + 1]} }
process coordinator {
    while( true ) {
        < await( count = n ) >
        ## ∀i, c[i] = min(c)
        count := 0; } }
```

Here *count* represents $counts[k]$ until the coordinator resets it.

Problems:

1. Still need an atomic increment
2. Unless all threads 'notice' $count = 0$ at 'about' the same time: deadlock.

Solutions

1. Distribute the representation of count among the threads.
   * Data refine count with an array of 0's and 1's that sums to count:
   $$count = \sum_{i=0}^{n-1} arrive[i]$$
2. Tell threads that round is over one at a time.
   * Use a separate flag to inform each thread
   $$continue[i] \Rightarrow c[i] = \min(c)$$

Coordinator

Global invariants: $count = \sum_{i=0}^{n-1} arrive[i]$ and $continue[i] \Rightarrow c[i] = \min(c)$, for all i

---

int $count := 0, c[n] = [(n)\ 0]$; — ghost variables
int $arrive[n] := ([n]0)$;
bool $continue[n] := ([n]false)$;

process Worker[$i = 0$ **to** $n - 1$] {
    while (*true*) {
        code to implement task i;
        $\langle arrive[i] := 1; count \mathrel{+}= 1; c[i] \mathrel{+}= 1; \rangle$
        $\langle$**await**$(continue[i])\rangle$
        $continue[i] := false;$ } }

process Coordinator {
    while (**true**) {
        for [$i = 0$ **to** $n - 1$] $\langle$**await**$(arrive[i] = 1)\rangle$
        ## $count = n$ (and $\therefore \forall i, c[i] = \min(c)$)
        for [$i = 0$ **to** $n-1$] $\langle arrive[i] := 0; count \mathrel{-}= 1; \rangle$
        ## $count = 0 \wedge \forall i, c[i] = \min(c)$
        for [$i = 0$ **to** $n - 1$] $continue[i] := true;$ } }

Using only arrive

Global invariants: $count = \sum_{i=0}^{n-1} arrive[i]$ and $arrive[i] = 0 \Rightarrow c[i] = \min(c)$, for all i

---

int $count := 0, c[n] = [(n)\ 0]$; — ghost variables
int $arrive[n] := ([n]0)$;

process Worker[$i = 0$ **to** $n - 1$] {
    while (*true*) {
        code to implement task i;
        $\langle arrive[i] := 1; count \mathrel{+}= 1; c[i] \mathrel{+}= 1; \rangle$
        $\langle \mathbf{await}(arrive[i] = 0) \rangle$ } }

process Coordinator {
    while (**true**) {
        for [$i = 0$ **to** $n - 1$] $\langle \mathbf{await}(arrive[i] = 1) \rangle$
        ## $count = n$ (and $\therefore \forall i, c[i] = \min(c)$)
        for [$i = 0$ **to** $n-1$] $\langle arrive[i] := 0; count \mathrel{-}= 1; \rangle$
    } }

Flag Synchronization Principles

- A thread that waits for a synchronization flag to be set should be the one to clear the flag.
- A flag should not be set until it is known to be clear.

Deadlock

- Exercise: show that neither of the last two algorithms can deadlock.

Inefficiencies

- Extra thread for `Coordinator`
- `Coordinator` is bottleneck.
- **Solutions**
  - Combining Tree Barrier
  - Symmetric Barrier

Asymmetric 2-thread barrier — abstract version

Global invariants

$s[i] \leq c[j] + 1$, for all i and j (desired invariant)
$c[i] \leq s[i] \leq c[i] + 1$, for all i (locally true)
$c[r] \leq c[l]$ (asymmetry)

---

int $c[2] := [0, 0], s[2] := [0, 0]$;

Root repeats:	Leaf repeats:
$s[r] \mathrel{+}= 1$; Round $\langle \mathbf{await}(c[r] < c[l]) \rangle$ ## $c[r] < c[l]$ $c[r] \mathrel{+}= 1;$ ## $c[r] = c[l]$	## $c[r] = c[l]$ $s[l] \mathrel{+}= 1$; Round $c[l] \mathrel{+}= 1$; ## $c[r] < c[l]$ $\langle \mathbf{await}(c[r] = c[l]) \rangle$

Asymmetric 2-thread barrier — state version

as before +

$$q = 0 \Rightarrow c[r] = c[l]$$
$$q = 1 \Rightarrow c[r] < c[l]$$

---

int $c[2] := [0,0], s[2] := [0,0], q := 0$;

Root repeats:	Leaf repeats:
$s[r]$ += 1 ; Round $\langle$**await**($q = 1$)$\rangle$ ## $q = 1$ $< c[r]$ += $1; q := 0; >$	## $q = 0$ $s[l]$ += 1 ; Round $\langle c[l]$ += $1; q := 1; \rangle$ $\langle$**await**($q = 0$)$\rangle$ ## $q = 0$

Asymmetric 2-thread barrier — flag version

as before +

$$\begin{aligned} arrive &\Rightarrow q = 1 \\ continue &\Rightarrow q = 0 \\ q = 2 &\Rightarrow c[r] = c[l] \end{aligned}$$

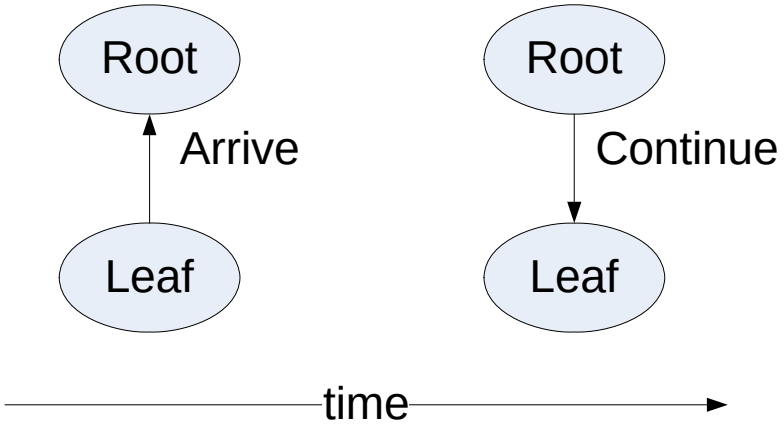
---

int $c[2] := [0,0], s[2] := [0,0], q := 0$;
boolean $arrive := \mathbf{false}, continue := \mathbf{false};$

Root repeats:	Leaf repeats:
$s[r]$ += 1 ; Round $\langle \mathbf{await}(\ arrive\ )\rangle$ ## $q = 1$ $\langle c[r] += 1; q := 2;$ $arrive := \mathbf{false};\rangle$ ## $q = 2$ $\langle continue := \mathbf{true}; q := 0;\rangle$	## $q = 0$ $s[l] += 1;$ Round $\langle c[l] += 1; q := 1;$ $arrive := \mathbf{true};\rangle$ $\langle \mathbf{await}(\ continue\ )\rangle$ $continue := \mathbf{false};$

Asymmetric 2-thread barrier — no ghosts

2 threads. One is the “root” one is the “leaf”



---

boolean $arrive := \mathbf{false}, continue := \mathbf{false};$

Root repeats:	Leaf repeats:
Round $\langle \mathbf{await}(arrive) \rangle$ $arrive := false;$ $continue := \mathbf{true};$	Round $arrive := \mathbf{true};$ $\langle \mathbf{await}(continue) \rangle$ $continue := \mathbf{false};$

Combining tree barrier

Threads are arranged in a binary tree.

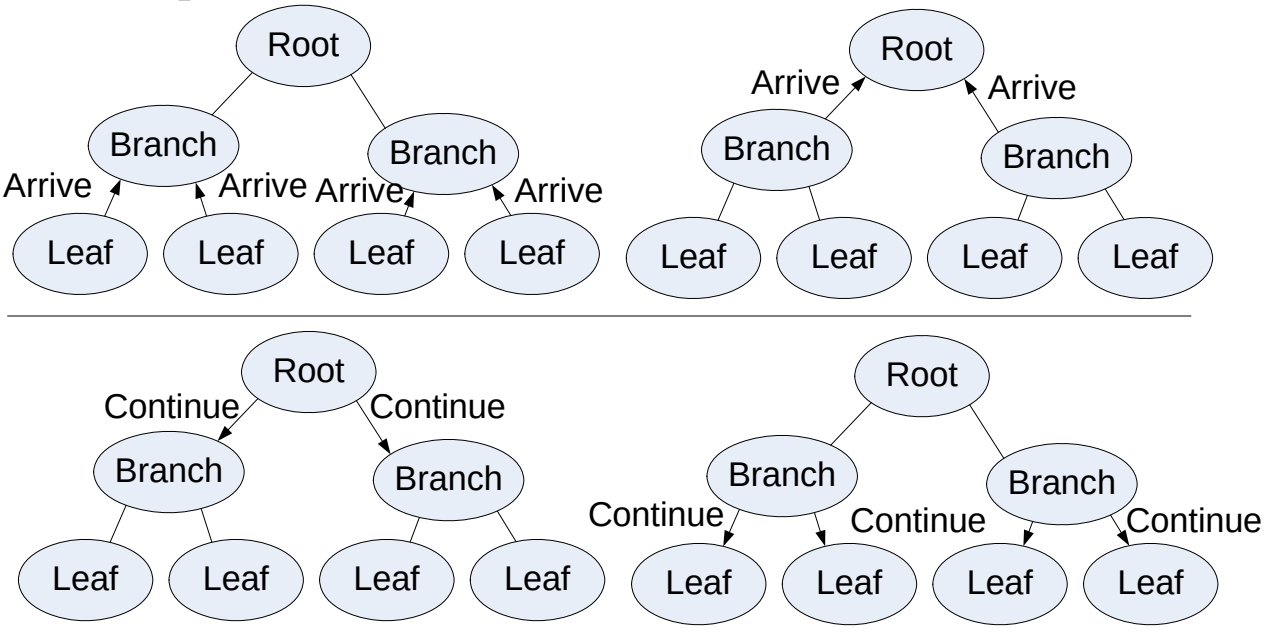
Each node waits for its children to tell it they are at the barrier.

Then it tells its parent that it, and thus its children, have reached the barrier.

Until the root knows that all threads have reached the barrier.

Then each parent tells its children to proceed.

And then proceeds itself.



Symmetric barrier

Idea

- Every thread does the same number of steps.
- Threads tell others that they are done and that other threads are done, until all threads know that all others are done

Two thread barrier

---

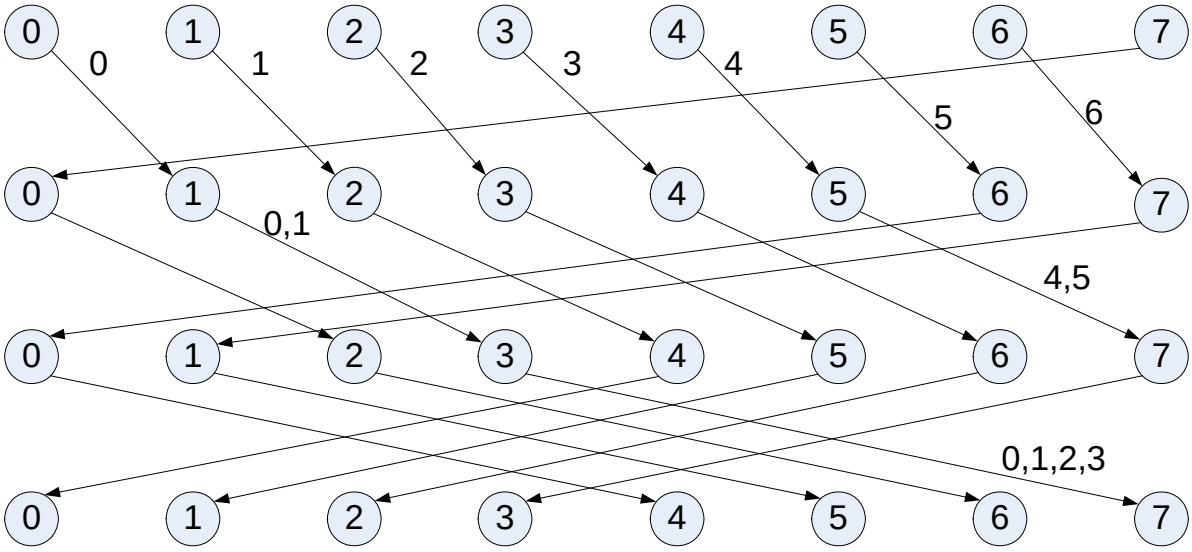
init: **int** $arrive[2] := [0, 0];$

# Thread i	# Thread j
$\langle$**await**$(\ arrive[i] = 0)\rangle$ $arrive[i] := 1;$ $\langle$**await**$(\ arrive[j] = 1)\ \rangle$ $arrive[j] := 0;$	$\langle$**await**$(\ arrive[j] = 0)\rangle$ $arrive[j] := 1;$ $\langle$**await**$(\ arrive[i] = 1)\rangle$ $arrive[i] := 0;$

Multiple thread version.

For 2^k threads each thread engages in k stages, each similar to the above. At each stage it sends information about itself and about all other threads it "knows about" to another thread.

- After m stages, the thread 'knows' that 2^m threads have reached the barrier.
- Thread i sends information to thread $i \oplus 2^m$ in stage m. (counting from 0).
- ($i \oplus 2^m$ means $(i + 2^m) \bmod n$)



Multiple thread. Implementation

Count the number stages passed.

$\ominus$ is subtraction mod n; k is $\lceil \log_2(n) \rceil$

Global invariant:

for all $r \in \mathbb{N}$, $j \in \{0, 1, ..., n-1\}$, $p \in \left\{0, 1, ..., 2^{r \bmod k} - 1\right\}$,

$$arrive[j] \geq r \Rightarrow c[j \ominus p] \geq \lfloor r/k \rfloor$$

---

init: **int** $arrive[i] := [(n)k - 1]$;
Thread i repeats
$s[i] \mathrel{+}= 1$;
round
$c[i] \mathrel{+}= 1$;
int $m := 0$;
Inv. $(\forall p \in \{0.1, ...2^m - 1\} \cdot c[i \ominus p] \geq c[i])$
while($m < k$ **) {**
    $arrive[i] \mathrel{+}= 1$;
    ## $arrive[i] = c[i] \times k + m$
    $\langle$**await(** $arrive[i \ominus 2^m] \geq arrive[i]$**)**$\rangle$
    $m \mathrel{+}= 1$; **}**

Needs infinite capacity ints, though.

Data Parallel Algorithms

Parallel Prefix: $\forall i \cdot 0 \leq i < n \Rightarrow sum[i] = \sum_{j=0}^{i} a[j]$

---

int $a[n], sum[n], old[n]$;
process Sum[$i = 0$ **to** $n - 1$] {
    int $d := 1$; # distance
    $sum[i] := a[i]$; # initialize to a
    $barrier(i)$;
    while ($d < n$) {
        $old[i] := sum[i]$;
        $barrier(i)$;
        if ($(i - d) >= 0$) $sum[i] \mathrel{+}= old[i - d]$;
        $barrier(i)$;
        $d \mathrel{+}= d$; } } # double the distance

Jacobi Iteration (Laplace's eqn):

```
real grid[0:n+1,0:n+1], newgrid[0:n+1,0:n+1];
bool converged := false;
process Grid[i = 1 to n, j = 1 to n] {
        while (!converged) {
                newgrid[i,j] :=
                (grid[i-1,j] + grid[i+1,j] +
                grid[i,j-1] + grid[i,j+1]) / 4;
                <converged := (test for convergence);>
                barrier( );
                grid[i,j] := newgrid[i,j];
                barrier( ); } }
```

Bag of Tasks

```
while ( there are more tasks to do ) {
        get task from the bag;
        execute the task, possibly
        generating new ones;
        }
```

- Task is independent unit of work.
- Bag represents collection of tasks.
- Scalable — set number of workers to number of processors.
- Load balanced — if a tasks takes longer, other workers will do more tasks.

Example: Adaptive Quadrature

```
process Worker[w = 1 to PR] {
    while( true ) {
        barrier ( ) ;
        double l, r, fl, fr, lrarea, m, fm, larea, rarea;
        while ( true ) {
            <idle ++ ; >
            < await (bag.size() > 0 || idle == PR)
                if( idle==PR ) break ;
                bag.get((l, r, fl, fr, lrarea)) >
            < idle--; >
            m := (l+r)/2;       fm := f(m);
            larea := (fl + fm) * (m - l) / 2;
            rarea := (fm + fr) * (f - m) / 2;
            if( abs(larea+rarea - lrarea) > EPS) {
                <bag.put((l, m, fl, fm, larea));>
                <bag.put(((m, r, fm, fr, rarea));> }
            else {
                <total += larea+rarea;> } }
        barrier( ); } }
```

```
process Coordinator() {
    while( true ) {
        idle := 0 ;
        total := 0 ;
        put the next top level task in the bag
        barrier( ) ;
        assert the bag is empty
        output total
        barrier( ) ; } }
```