Privatization

Privatization means making private copies of a shared variable.

What is the advantage of privatization?

Of the three kinds of variables in the table above, which kind(s) does it make sense to privatize?

Under what conditions is a variable privatizable?

- If it is always defined (=written) in program order by a task before use (=read) by the same task (Case 1).
- If its values in different parallel tasks are known ahead of time, allowing private copies to be initialized to the known values (Case 2).

When a variable is privatized, one private copy is made for each thread (not each task).

Result of privatization: R/W conflicting $\rightarrow$ R/W non-conflicting

Let's revisit the examples.

Example 1

With each **for** *i* iteration a separate task, which of the R/W conflicting variables are privatizable?

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Which case does each such variable fall into?

We can think of privatized variables as arrays, indexed by process ID:

Example 2

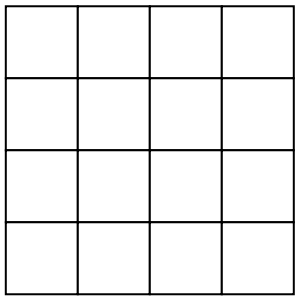
Parallel tasks: each **for** *j* loop iteration.

Is the R/W conflicting variable *j* privatizable? If so, which case does it represent?

Reduction

Reduction is another way to remove conflicts. It is based on partial sums.

Suppose we have a large matrix, and need to perform some operation on all of the elements—let's say, a sum of products—to produce a single result.



We could have a single processor undertake this, but this is slow and does not make good use of the parallel machine.

So, we divide the matrix into portions, and have one processor work on each portion.

Then after the partial sums are complete, they are combined into a global sum. Thus, the array has been "reduced" to a single element.

Examples:

- addition (+), multiplication (*)
- Logical (**and**, **or**, …)

The *reduction variable* is the scalar variable that is the result of a reduction operation.

Criteria for reducibility:

- Reduction variable is updated by each task, and the order of update _____.
- Hence, the reduction operation must be _____ _____ .

Goal: Compute

$y = y\_init$ **op** $x1$ **op** $x2$ **op** $x3$ … **op** x_n

op is a reduction operator if it is *commutative*

u **op** $v = v$ **op** u

and *associative*

(u **op** v) **op** $w = u$ **op** (v **op** w)

Summary of scope criteria

Should be declared private	Should be declared shared	Should be declared reduction	Non-privatizable R/W conflicting

Example 1

with **for** *i* parallel tasks

Fill in the answers here.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Read-only	R/W non-conflicting	R/W conflicting

Declare as shared	Declare as private

Example 2

with **for** *j* parallel tasks

Fill in the answers here.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```

Read-only	R/W non-conflicting	R/W conflicting

Declare as shared	Declare as private

Example 3

Consider matrix multiplication.

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
    C[i][j] = 0.0;
    for (k=0; k<n; k++) {
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
  }
}
```

Exercise:
Suppose the parallel tasks are **for** *k* iterations. Determine which variables are conflicting, which should be declared as private, and which need to be protected against concurrent access by using a critical section.

Read-only	R/W non-conflicting	R/W conflicting

Declare as shared	Declare as private

Which variables, if any, need to be protected by a critical section?

Now, suppose the parallel tasks are **for** *i* iterations. Determine which variables are conflicting, which should be declared as private, and which need to be protected against concurrent access by using a critical section.

Read-only	R/W non-conflicting	R/W conflicting

Declare as shared	Declare as private

Which variables, if any, need to be protected by a critical section?

Synchronization

Synchronization is how programmers control the sequence of operations that are performed by parallel threads.

Three types of synchronization are in widespread use.

- *Point-to-point:*
  - a pair of *post*() and *wait*()
  - a pair of *send*() and *recv*() in message passing
- *Lock*
  - a pair of *lock*() and *unlock*()
  - only one thread is allowed to be in a locked region at a given time
  - ensures mutual exclusion
  - used, for example, to serialize accesses to R/W concurrent variables.

- *Barrier*
    - a point past which a thread is allowed to proceed only when all threads have reached that point.

Lock

What problem may arise here?

```
// inside a parallel region
for (i=start_iter; i<end_iter; i++)
  sum = sum + a[i];
```

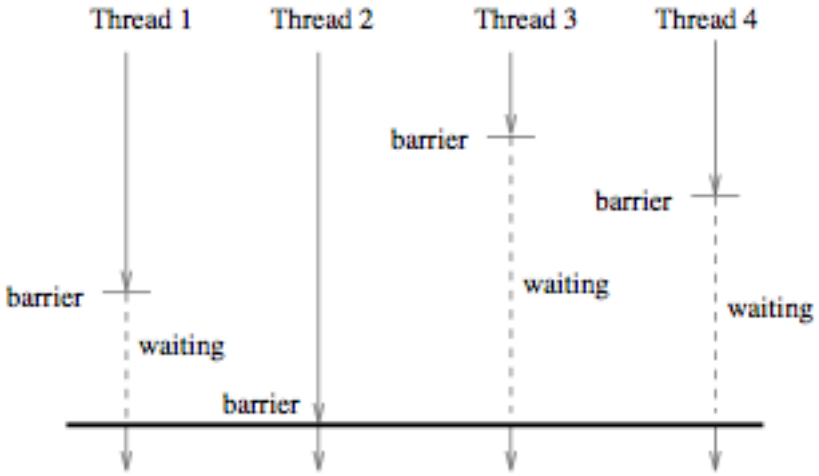
A lock prevents more than one thread from being inside the locked region.

```
// inside a parallel region
for (i=start_iter; i<end_iter; i++) {
    lock(x);
    sum = sum + a[i];
    unlock(x);
}
```

Issues:

- What granularity to lock?
- How to build a lock that is correct and fast.

Barrier: Global event synchronization



A barrier is used when the code that follows requires that all threads have gotten to this point. Example: Simulation that works in terms of timesteps.

Load balance is important.

Execution time is dependent on the slowest thread.

This is one reason for gang scheduling and avoiding time sharing and context switching.