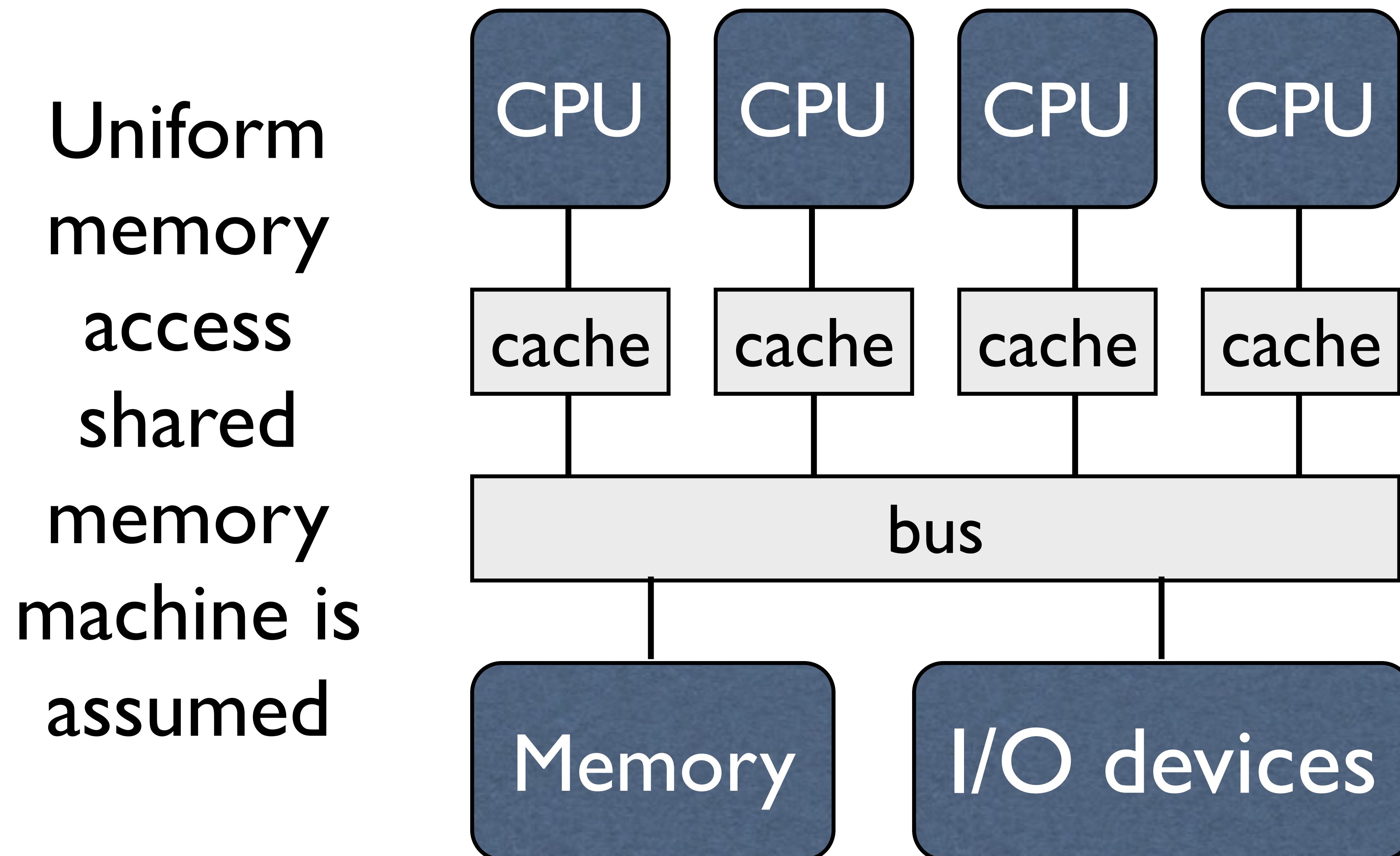
Basic OpenMP

What is OpenMP

- An open standard for shared memory programming in C/C++ and Fortran
- supported by Intel, Gnu, Microsoft, Apple, IBM, HP and others
- Compiler directives and library support
- OpenMP programs are typically still legal to execute sequentially
- **Allows program to be incrementally parallelized**
- Can be used with MPI -- will discuss that later

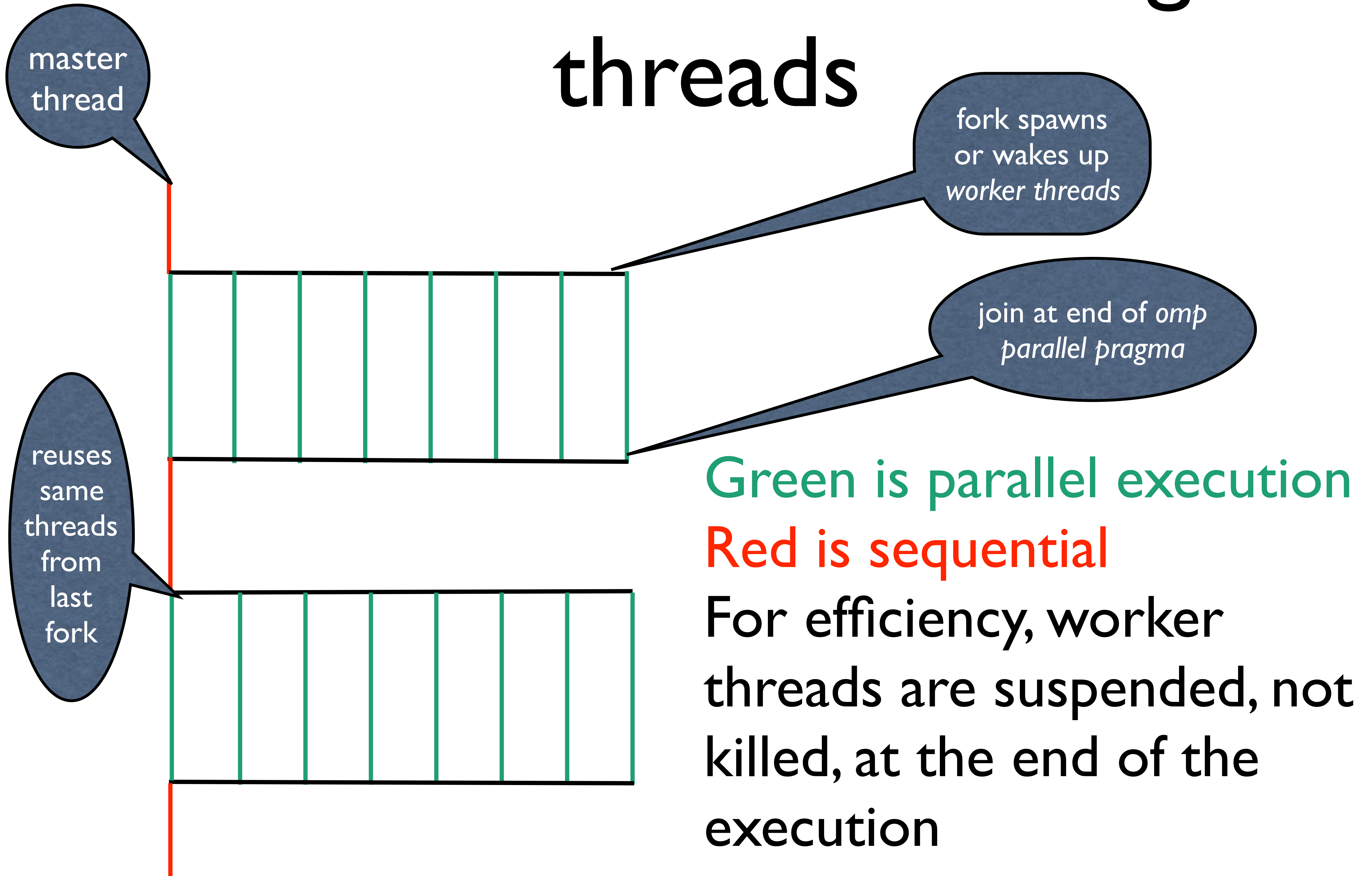
Basic OpenMP Hardware Model



Fork/Join Parallelism

- Program execution starts with a single *master thread*
- Master thread executes sequential code
- When parallel part of the program is encountered, a *fork* utilizes other *worker* threads
- At the end of the parallel region, a *join* kills or suspends the worker threads

Parallel execution using threads



Where is the work in programs?

- For many programs, most of the work is in loops
- C and Fortran often use loops to express *data parallel* operations
  - the same operation applied to many independent data elements

```
for (i = first; i < size; i += prime)
  marked[i] = 1;
```

What can run in parallel?

Consider the loop:

```
for (i=1; i<n; i++) {
  a[i] = b[i] + c[i];
  c[i] = a[i-1]
}
```

Let each iteration execute in parallel with all other iterations on its own processor

time

i = 1	*i = 2*	*i = 3*
a[1] = b[1] + c[1]	a[2] = b[2] + c[2]	a[3] = b[3] + c[3]
c[1] = a[0]	c[1] = a[0]	c[3] = a[2]

Note that data is produced in one iteration and consumed in another.

What can run in parallel?

cores or processors

Consider the loop:

```
for (i=1; i<n; i++) {
    a[i] = b[i] + c[i];
    c[i] = a[i-1]
}
```

time

i = 3
a[3] = b[3] + c[3]
c[3] = a[2]

i = 2
a[2] = b[2] + c[2]
c[1] = a[0]

What if the processor executing iteration i=2 is delayed for some reason? *Disaster* - the value of a[2] to be read by iteration i=3 is not ready when the read occurs!

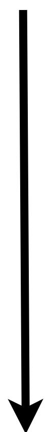
Cross-iteration *dependences*

Consider the loop:

```
for (i=1; i<n; i++) {
  a[i] = b[i] + c[i];
  c[i] = a[i-1]
}
```

Orderings that must be enforced to ensure the correct order of reads and writes are called *dependences*.

time



i = 1
a[1] = b[1] + c[1]
c[1] = a[0]

i = 2
a[2] = b[2] + c[2]
c[1] = a[0]

i = 3
a[3] = b[3] + c[3]
c[3] = a[2]

A dependence that goes from one iteration to another is a *cross iteration*, or *loop carried* dependence

Cross-iteration dependences

Consider the loop:

```
for (i=1; i<n; i++) {
   a[i] = b[i] + c[i];
   c[i] = a[i-1]
}
```

Loops with cross iteration dependences cannot be executed in parallel unless mechanisms are in place to ensure dependences are honored.

time

i = 1	*i = 2*	*i = 3*
a[1] = b[1] + c[1]	a[2] = b[2] + c[2]	a[3] = b[3] + c[3]
c[1] = a[0]	c[1] = a[0]	c[3] = a[2]

We will generally refer to a loop as *serial* or *not parallelizable* if dependences do not span the code that is to be run in parallel.

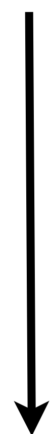
Cross-iteration dependences

Consider the loop:

```
for (i=1; i<n; i++) {
  a[i] = b[i] + c[i];
  c[i] = a[i]
}
```

Loops without cross iteration dependences can run in parallel, because out-of-order execution of iterations doesn't affect what is read or written in an iteration.

time



i = 3
a[3] = b[3] + c[3]
c[3] = a[3]

i = 1
a[1] = b[1] + c[1]
c[1] = a[1]

i = 2
a[2] = b[2] + c[2]
c[2] = a[2]

We will generally refer to a loop as *parallel* or *parallelizable* if dependences do not span the code that is to be run in parallel.

Where is parallelism found?

- Most work in most programs, especially numerical programs, is in a loop
- Thus effective parallelization generally requires parallelizing loops
- *Amdahl's law* (discussed later in the course) says that, for example, if we parallelize 90% of a program we will get, at most, a *speedup* of 10X, 99% a speedup of 100X. To effectively utilize 1000s of processors, we need to parallelize 99.9% or more of a program!

OpenMP *Pragmas*

- OpenMP expresses parallelism and other information using *pragmas*
- A C/C++ or Fortran compiler is free to ignore a pragma -- this means that OpenMP programs have serial as well as parallel semantics
  - outcome of the program should be the same in either case
- #pragma omp *<rest of the pragma>* is the general form of a pragma

pragma for parallel for

- OpenMP programmers use the *parallel for* pragma to tell the compiler a loop is parallel

```
#pragma omp parallel for
for (i=0; i < n; i++) {
   a[i] = b[i] + c[i];
```

Syntax of the *parallel for* control clause

for (index = *start;* index *rel-op val*; *incr)*

- *start* is an integer index variable
- *rel-op* is one of {<, <=, >=, >}
- *val* is an integer expression
- *incr* is one of {index++, ++index, index--, --index, index+=*val*, index-=*val*, index=index+*val,* index=*val*+index, index=index-*val*
- OpenMP needs enough information from the loop to run the loop on multiple threads

Each thread has an execution context

shared among threads
private to a thread

- The execution context contains
  - static and global variables - shared
  - heap allocated storage - shared
  - variables on the stack belonging to functions called along the way to invoking the thread - shared
  - a thread-local stack for functions invoked and block entered during the thread execution - private
- Each thread must be able to access all of the storage it references

Example of context

Consider the program below:

```
int v1;
...
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   ...
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for
   for (int i=0; i < n; i++) {
      int v4;
      T1 *v5 = malloc(sizeof(T1));
}}
```

Variables v1, v2, v3 and v4, as well as heap allocated storage, are part of the context of the `parallel for`.

Context before call to f1

Storage, assuming two threads
red is shared,
green is private to thread 0,
blue is private to thread 1

```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for
    for (int i=0; i < n; i++) {
        int v4;
        T1 *v5 = malloc(sizeof(T1));
}}
```

Context right after call to f1

Storage, assuming two threads

red is shared,

green is private to thread 0,

blue is private to thread 1

```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for
    for (int i=0; i < n; i++) {
        int v4;
        T2 *v5 = malloc(sizeof(T2));
}}
```

statics and globals: v1

global stack

main: v2

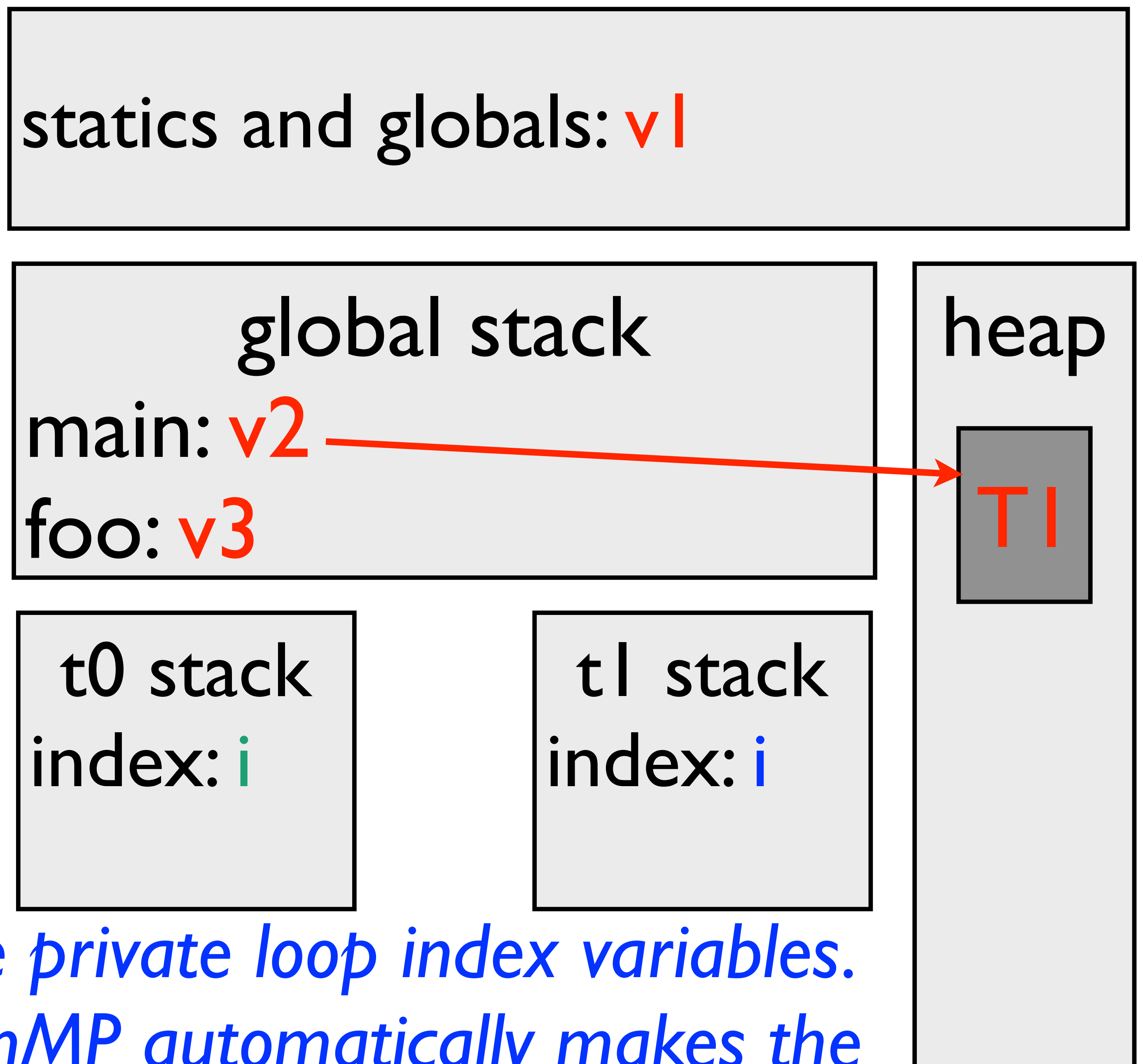
foo: v3

heap

Context at start of parallel for

Storage, assuming two threads
red is shared,
green is private to thread 0,
blue is private to thread 1

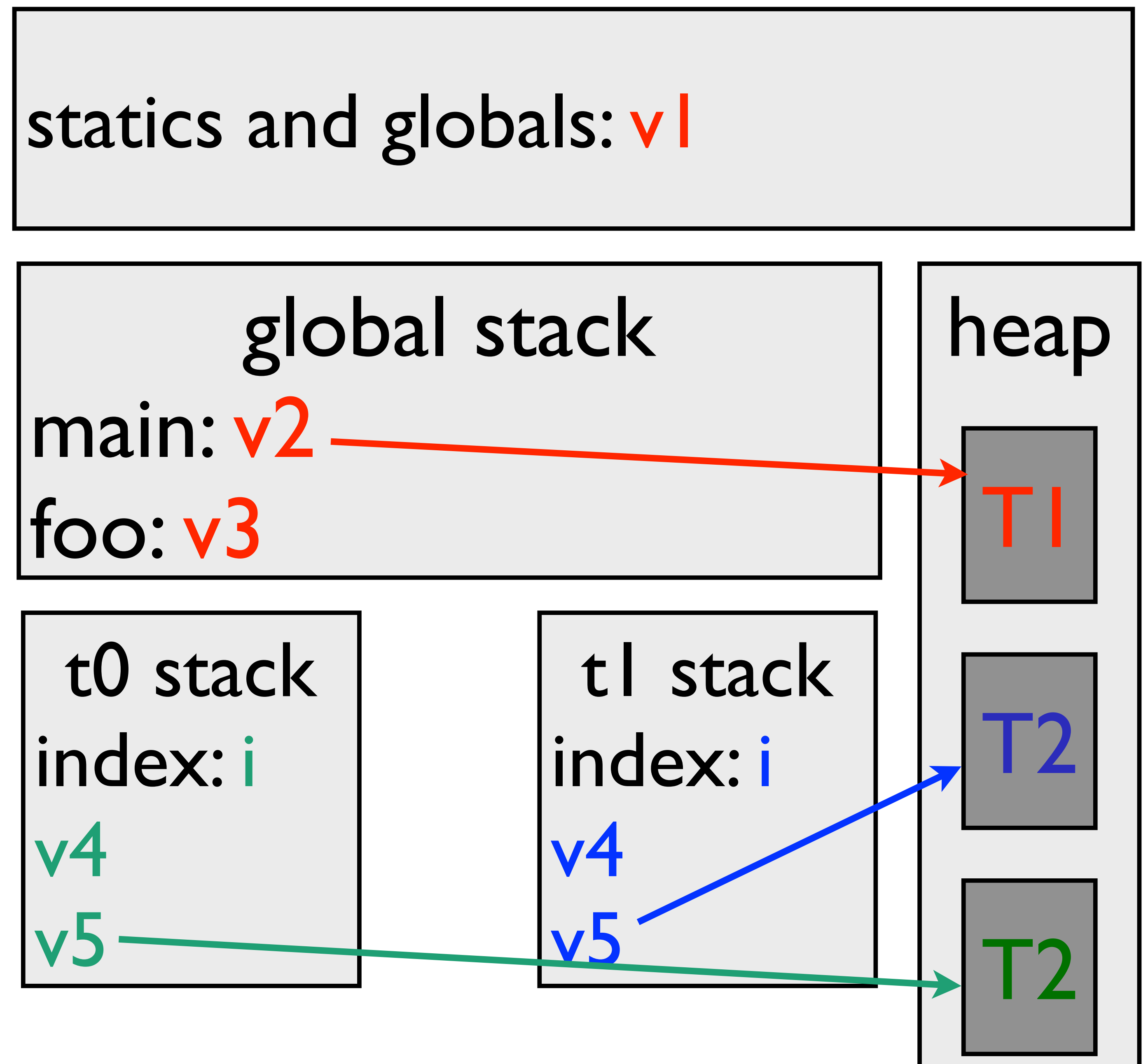
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for
    for (int i=0; i < n; i++) {
        int v4;
        T1 *v5 = malloc(sizeof(T1));
}}
```



Context after first iteration of the *parallel for*

Storage, assuming two threads
red is shared,
green is private to thread 0,
blue is private to thread 1

```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for
    for (i=0; i < n; i++) {
        int v4;
        T1 *v5 = malloc(sizeof(T1));
}}
```



Context after *parallel for* finishes

Storage, assuming two threads
red is shared,
green is private to thread 0,
blue is private to thread 1

```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for

    for (i=0; i < n; i++) {
        int v4;
        T1 *v5 = malloc(sizeof(T1));
}}
```

statics and globals: v1

global stack

main: v2

foo: v3

heap

T1

T2

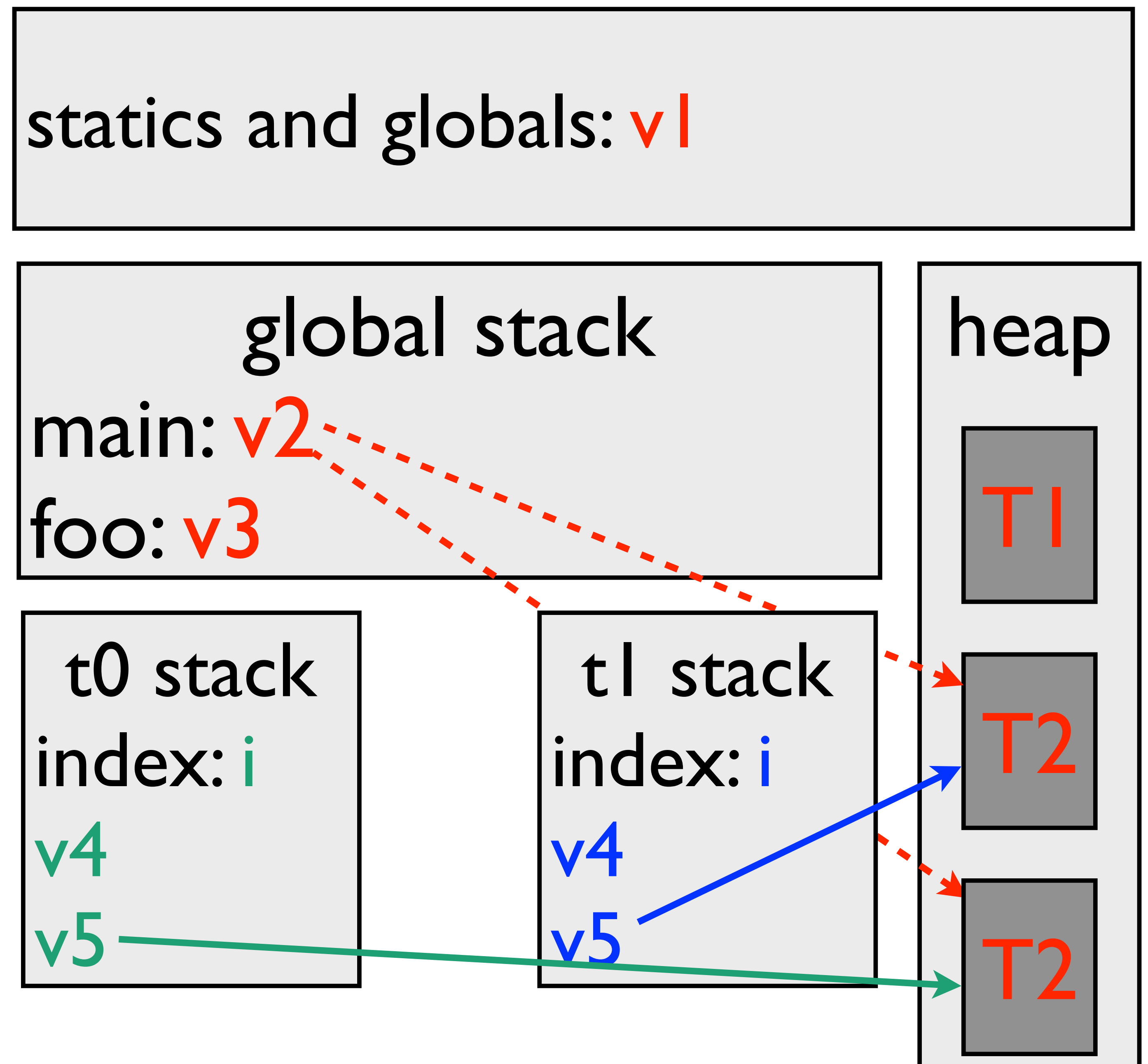
T2

A slightly different example -- after each thread has run at least *1* iteration

v2 points to one of the T2 objects that was allocated.

Which one? It depends.

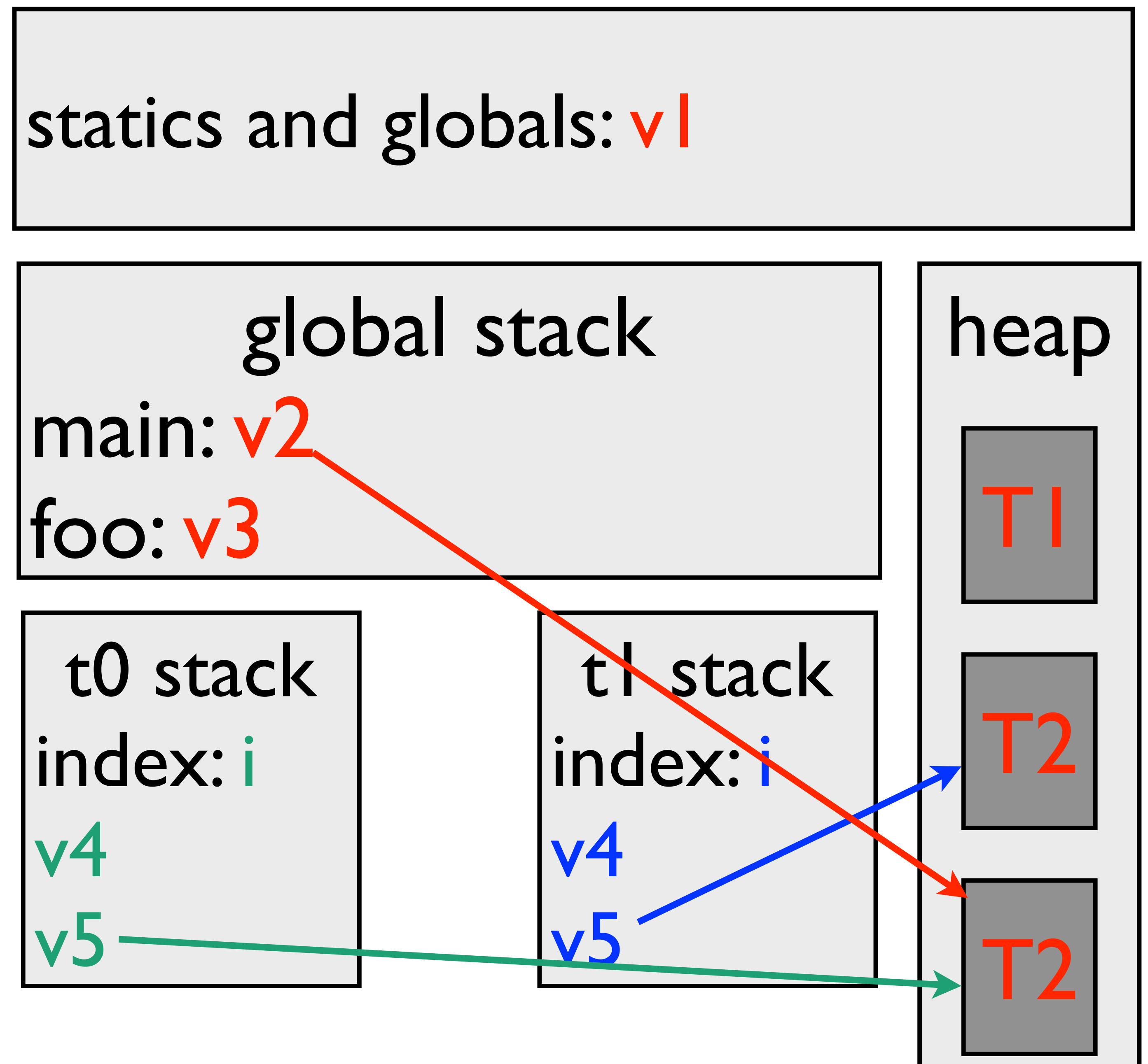
```
int v1;
...
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   ...
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for

   for (i=0; i < n; i++) {
      int v4;
      T2 *v5 = malloc(sizeof(T2));
      v2 = (T1) v5
}}
```



After each thread has run at least *1* iteration

v2 points to the T2 allocated by t0 if t0 executes the statement v2=(T1) v5; last

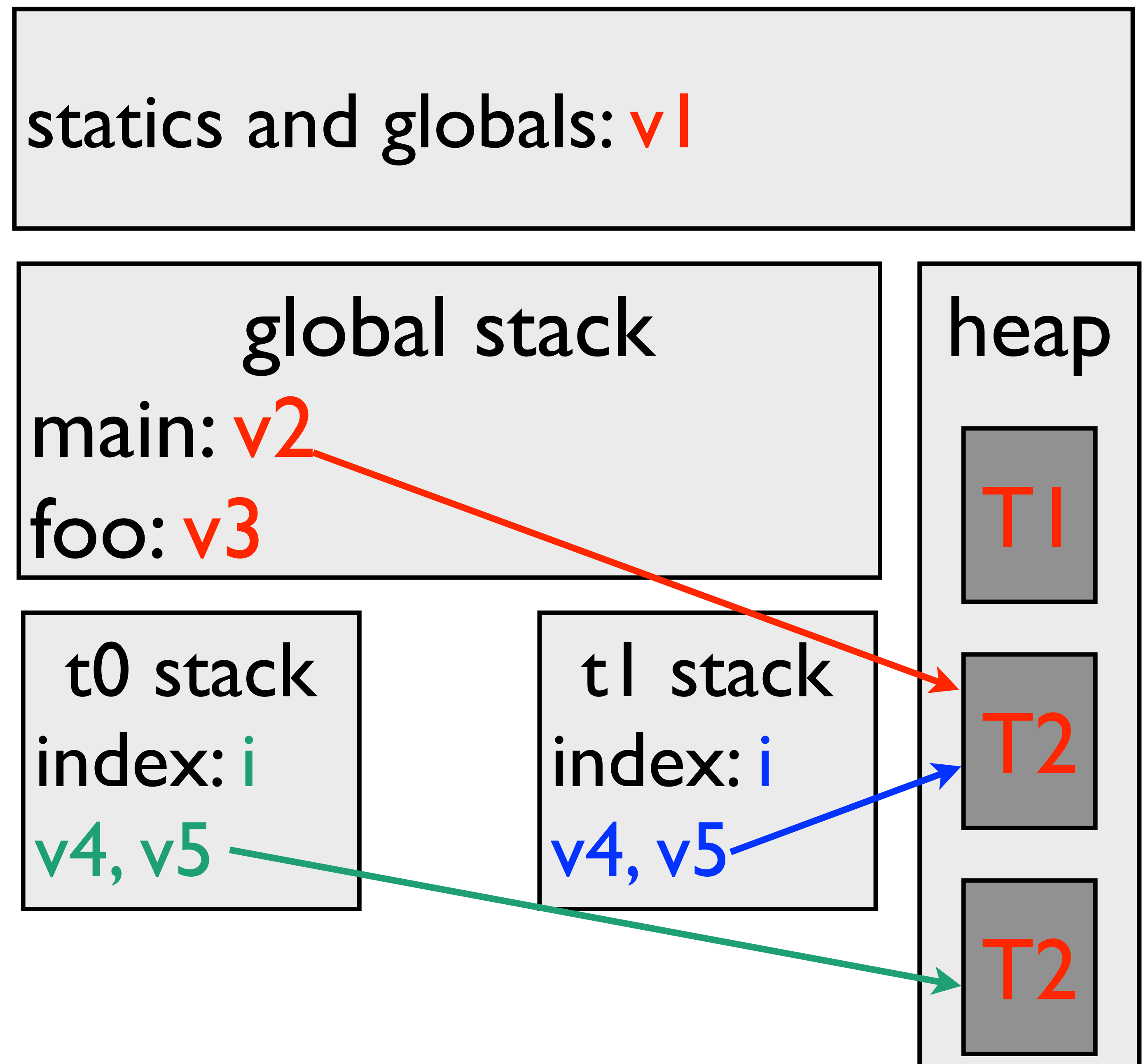
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for

    for (i=0; i < n; i++) {
        int v4;
        T2 *v5 = malloc(sizeof(T2));
        v2 = (T1) v5
}}
```



After each thread has run at least *1* iteration

v2 points to the T2 allocated by t1 if t1 executes the statement v2=(T1) v5; last

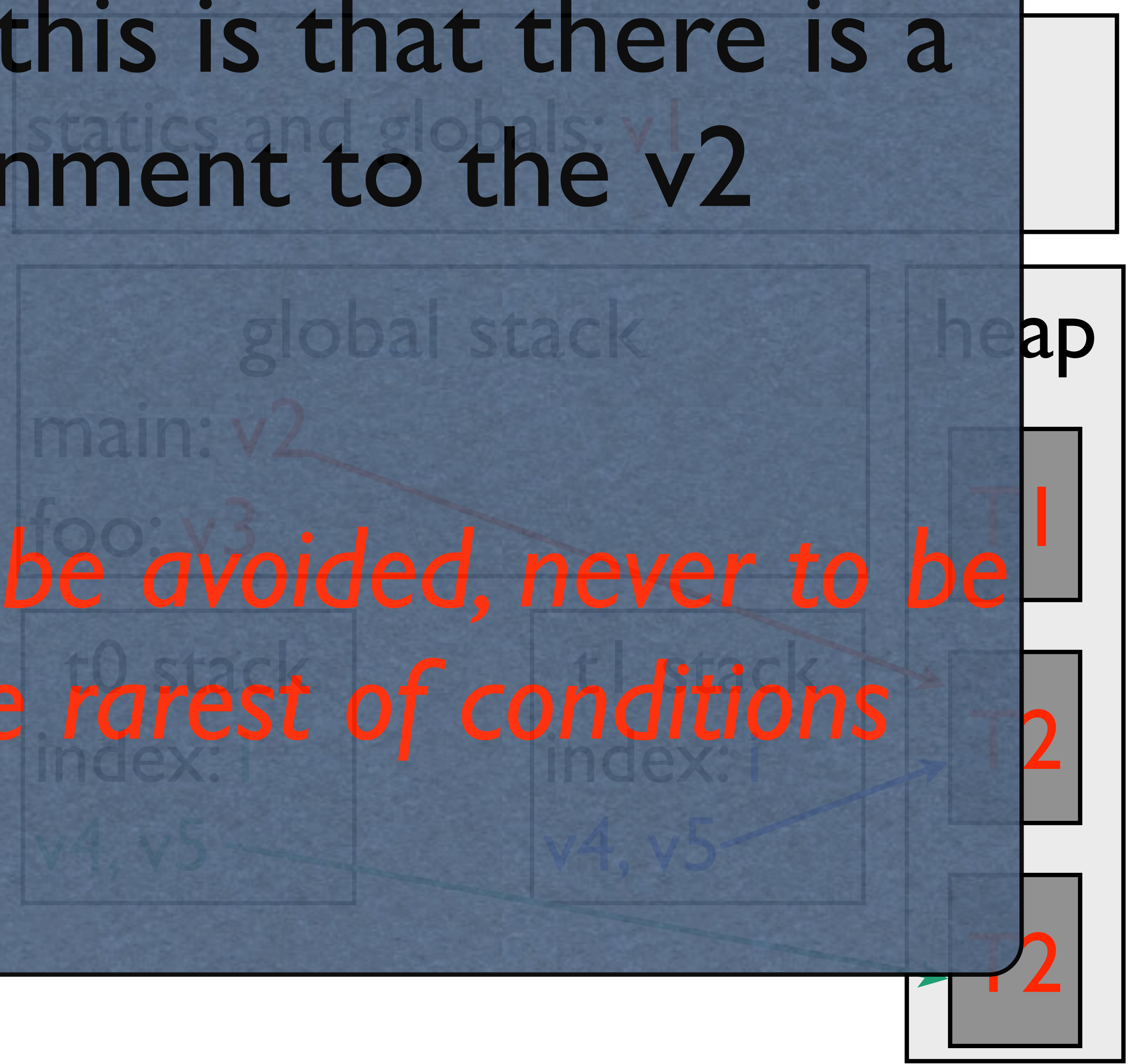
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for
    for (i=0; i < n; i++) {
        int v4;
        T2 *v5 = malloc(sizeof(T2));
        v2 = (T1) v5
}}
```



A slightly different example -- after each thread has run at least 1 iteration

v2 points to the T2 allocated by t1 if t1 executes the statement v2=(T1) v5 last

```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for

    for (i=0; i < n; i++) {
        int v4;
        T2 *v5 = malloc(sizeof(T2));
        v2 = (T1) v5
}}
```



A problem with this is that there is a *race* on the assignment to the v2 variable

Races are bad, to be avoided, never to be done except in the rarest of conditions

Another problem with this code

There is a memory leak!

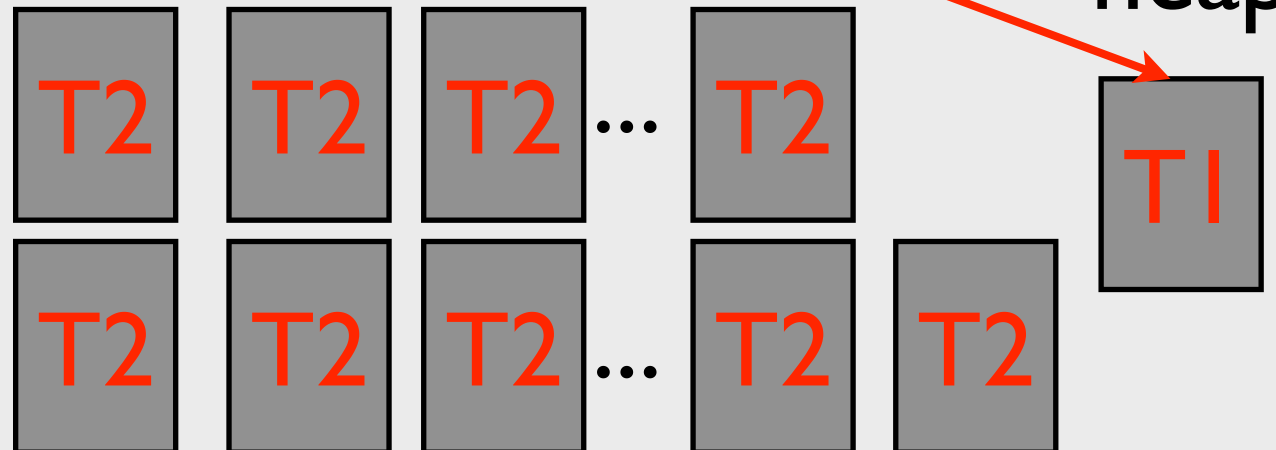
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for

    for (i=0; i < n; i++) {
        int v4;
        T1 *v5 = malloc(sizeof(T1));
}}
```

statics and globals: v1

global stack

main: v2

foo: v3



Querying the number of processors (really cores)

- Can query the number of physical processors
  - returns the number of *cores* on a multicore machine without *hyper threading*
  - returns the number of possible *hyperthreads* on a hyperthreaded machine

int omp_get_num_procs(void);

Setting the number of threads

- Number of threads can be more or less than the number of processors (cores)
  - if less, some processors or cores will be idle
  - if more, more than one thread will execute on a core/processor
    - Operating system and runtime will assign threads to cores
    - No guarantee same threads will always run on the same cores
- Default is number of threads equals number of cores controlled by the OS image (typically #cores on node/processor)

int omp_set_num_threads(int t);

Making more than the *parallel for* index private

```
int i, j;
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      a[i][j] = max(b[i][j],a[i][j]);
   }
}
```

Either the i or the j loop can run in parallel.

We prefer the outer i loop, because there are fewer parallel loop starts and stops.

Forks and joins are serializing, and we know what that does to performance.

Making more than the parallel for index private

```
int i, j;
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      a[i][j] = max(b[i][j],a[i][j]);
   }
}
```

Either the *i* or the *j* loop can run in parallel.

To make the *i* loop parallel we need to make *j* private.

Why? Because otherwise there is a *race* on *j*! Different threads will be incrementing the same *j* index!

Making the *j* index private

- *clauses* are optional parts of pragmas
- The *private* clause can be used to make variables private
- private (<*variable list*>)

```
int i, j;
#pragma omp parallel for private(j)
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      a[i][j] = max(b[i][j],a[i][j]);
   }
}
```

When is private needed?

- If a variable is declared in a parallel construct (e.g., a *parallel for*) no *private* is needed.
- Loop indices of *parallel for* is private by default.

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
   for (int j=0; j<n; j++) {
      a[i][j] = max(b[i][j],a[i][j]);
   }
}
```

j is private here because it is declared inside the parallel i loop

When is private needed?

- What if we want a variable that is private by default to be shared?
- Use the *shared* clause.

```
#pragma omp parallel for shared(t)
for (int i=0; i<n; i++) {
   int t;
   for (int j=0; j<n; j++) {
      a[i][j] = max(b[i][j],a[i][j]);
   }
}
```

Initialization of private variables

- use the *firstprivate* clause to give the private the value the variable with the same name, controlled by the master thread, had when the *parallel for* is entered.
- initialization happens once per thread, not once per iteration
- if a thread modifies the variable, its value in subsequent reads is the new value

```
double tmp = 52;
#pragma omp parallel for firstprivate(tmp)
for (i=0; i<n; i++) {
    tmp = max(tmp,a[i]);
}
```

tmp is initially 52 for all threads within the loop

Initialization of private variables

- What is the value at the end of the loop?

```
double tmp = 52;
#pragma omp parallel for firstprivate(tmp)
for (i=0; i<n; i++) {
    tmp = max(tmp,a[i]);
}
z = tmp;
```

Recovering the value of private variables from the last iteration of the loop

- use *lastprivate* to recover the last value written to the private variable in a sequential execution of the program
- *z* and *tmp* will have the value assigned in iteration $i = n\text{-}1$

```
double tmp = 52;
#pragma omp parallel for lastprivate(tmp) firstprivate(tmp)
for (i=0; i<n; i++) {
	tmp = max(tmp,a[i]);
}
z = tmp;
```

- note that the value saved by *lastprivate* will be the value the variable has in iteration $i=n\text{-}1$. What happens if a thread other than the one executing iteration $i=n\text{-}1$ found the max value?

Let's solve a problem

- Given an array *a* we would like the find the average of its elements
- A simple sequential program is shown below
- Our problem is to do this in parallel

```
for (i=0; i < n; i++) {
   t = t + a[i];
}
t = t/n
```

First (and wrong) try:

- Make *t* private
- initialize it to zero outside, and make it *firstprivate* and *lastprivate*
- Save the last value out

```
t = 0
#pragma omp parallel for firstprivate(t), lastprivate(t)
for (i=0; i < n; i++) {
   t += a[i];
}
t = t/n
```

What is wrong with this?

Second try:

```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
   t += a[i];
}
t = t/n
```

What is wrong with this?

Second try:

- Need to execute t+= a[i]; atomically
- Need to get the old value of `t`, add it to `a[i]`, and then save it to `t` without any other threads reading or writing `t` or `a[i]`.

```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
   t += a[i];
}
t = t/n
```

An example of atomic operations and why they are needed

ordering and *atomicity* are important and different

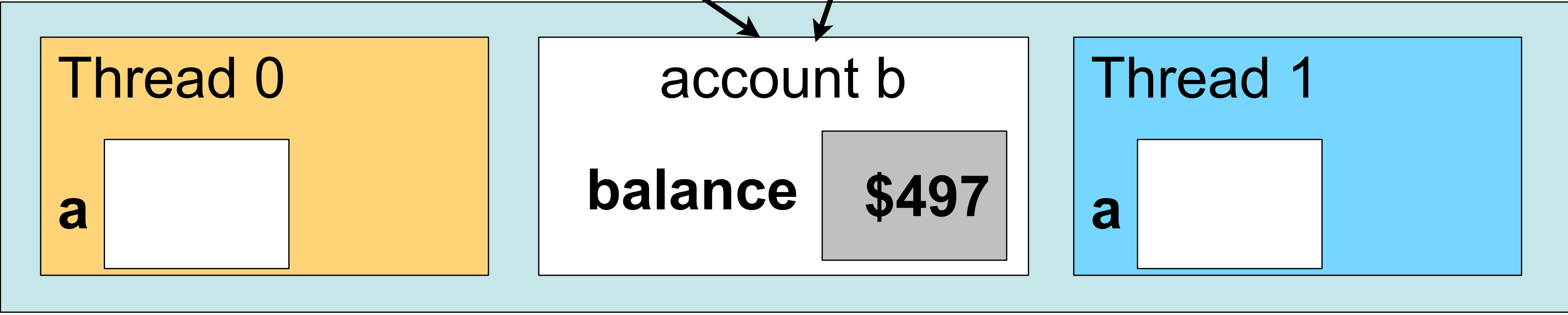
thread 0

thread 1

```
a = Balance;

a++;

Balance = a;
```

Both threads can access the same object

```
a = Balance;

a++;

Balance = a;
```



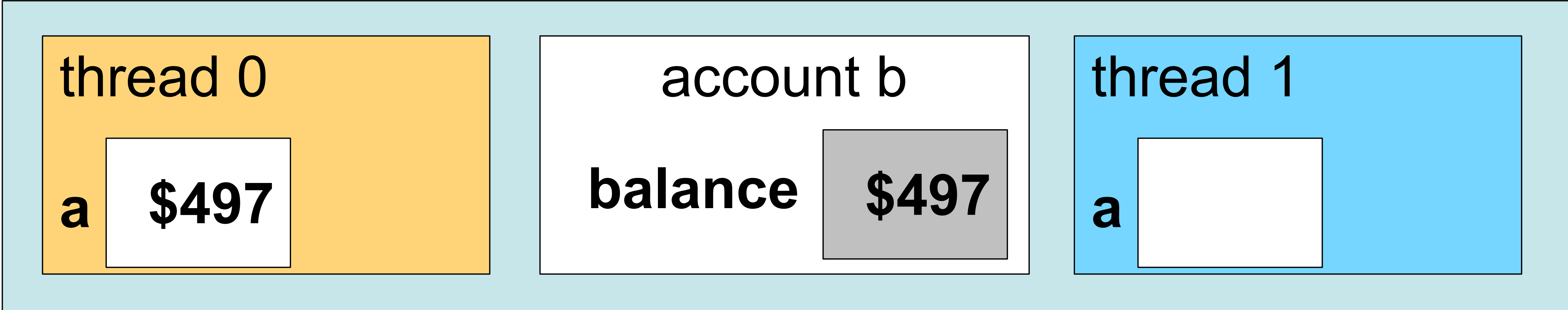
Program Memory

thread 0

```
a = Balance;

a++;

Balance = a;
```

thread 1

```
a = Balance;

a++;

Balance = a;
```



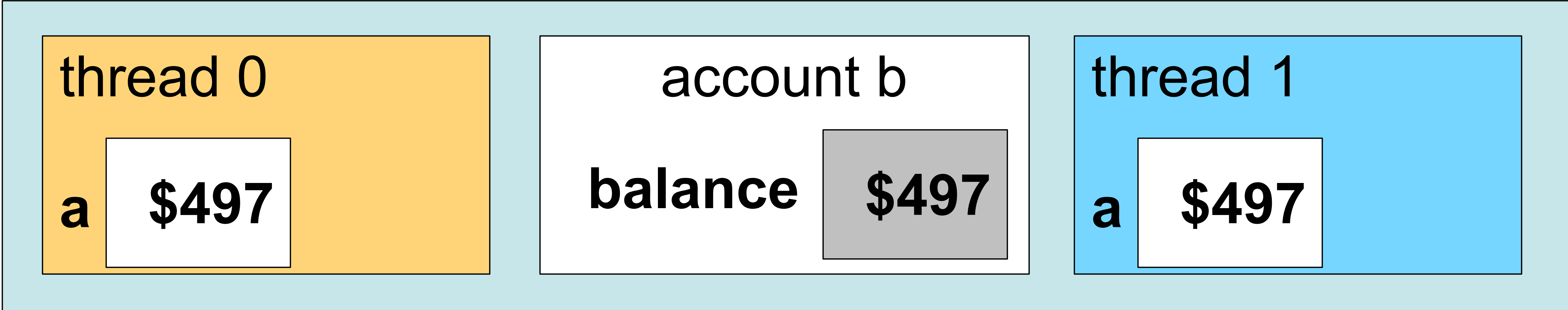
Program Memory

thread 0

```
a = Balance;

a++;

Balance = a;
```

thread 1

```
a = Balance;

a++;

Balance = a;
```

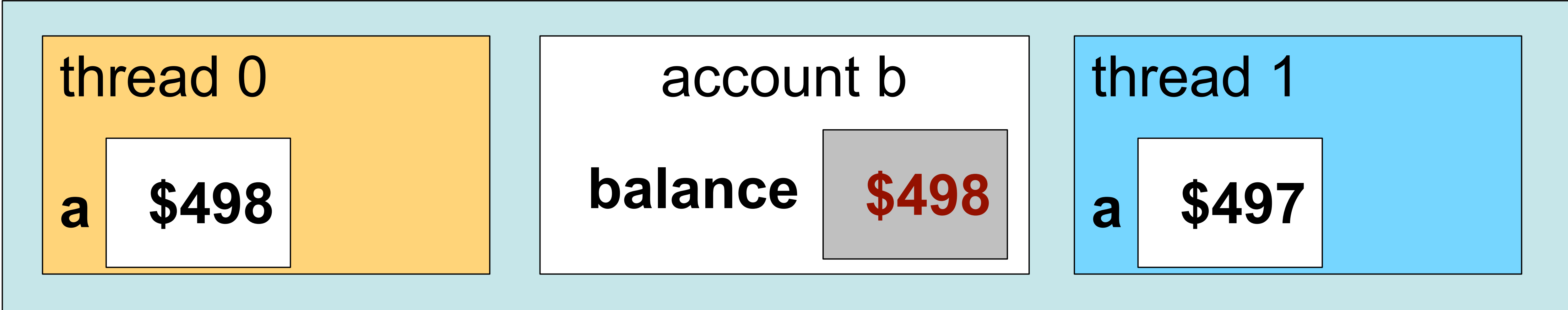


thread 0

```
a = Balance;

a++;

Balance = a;
```

thread 1

```
a = Balance;

a++;

Balance = a;
```



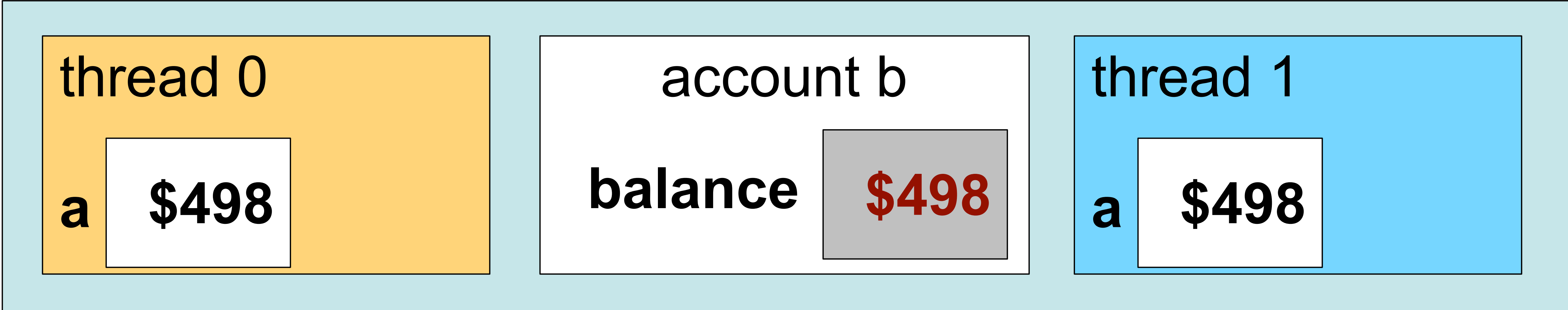
thread 0

```
a = Balance;

a++;

Balance = a;
```

The end result probably should have been $499. One update is lost.

thread 1

```
a = Balance;

a++;

Balance = a;
```



synchronization enforces atomicity

thread 0

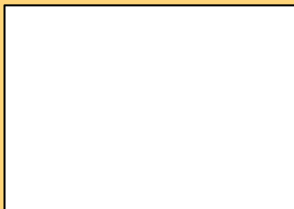
```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

Make them *atomic* using ***critical***

thread 1

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

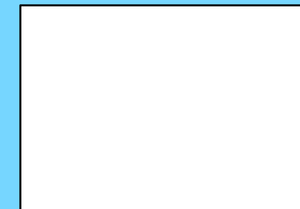
thread 0

a 

object b

balance **$497**

thread 1

a 

Program Memory

One thread acquires the lock

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

The other thread waits until the lock is free

thread 0	object b	thread 1
a	**balance** **$497**	**a**

One thread acquires the lock

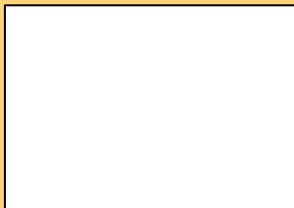
```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

The other thread waits until the lock is free

thread 0

a



object b

balance **$498**

thread 1

a **$498**

One thread acquires the lock

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

The other thread waits until the lock is free

thread 0	object b	thread 1
a **$498**	**balance** **$498**	**a** **$498**

One thread acquires the lock

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

The other thread waits until the lock is free

thread 0

a **$499**

object b

balance **$499**

thread 1

a **$498**

Locks typically do not enforce ordering

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

Either order is possible

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

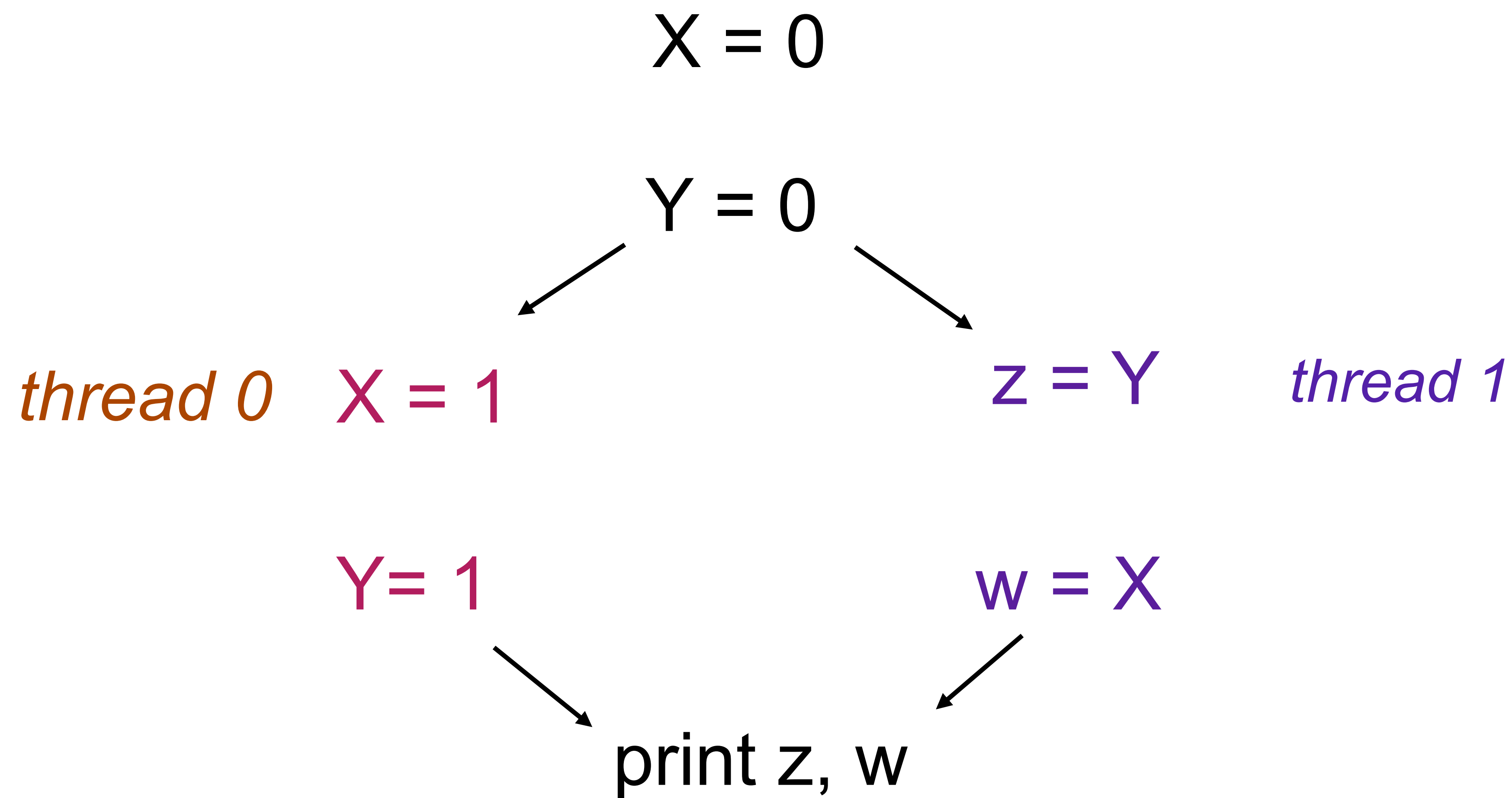
For many (but not all) programs, either order is correct

```
#omp critical
{
   a = Balance;
   a++;
   Balance = a;
}
```

Sequential Consistency (SC)

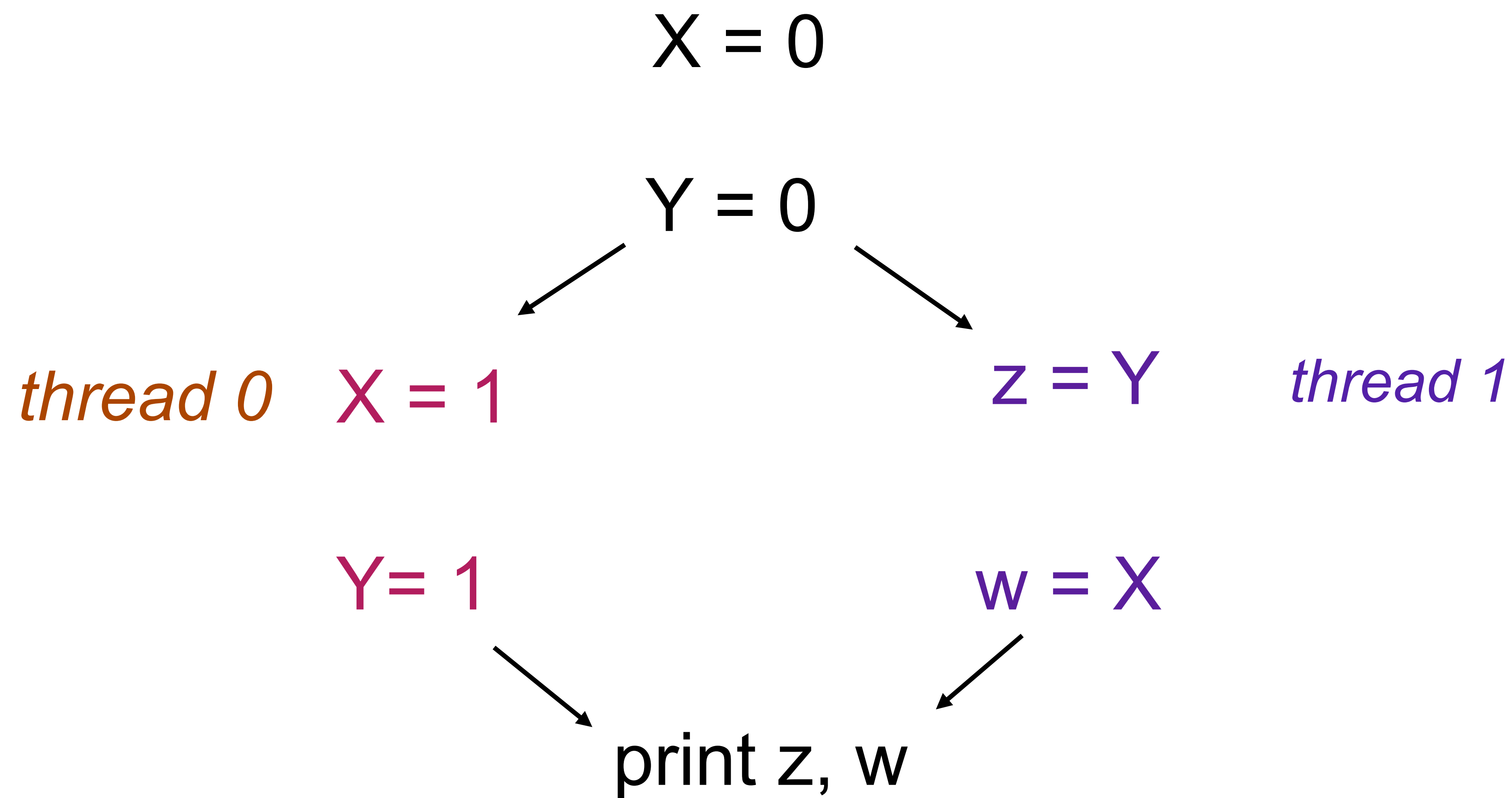
- Coherence says that a read will get the last value written for a variable
- Consistency is concerned with the interactions between writes to different variables
- Sequential consistency (see Lamport paper) is when ... *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

SC Example



Question: Is it legal for ***z == 1*** and ***w == 0***?

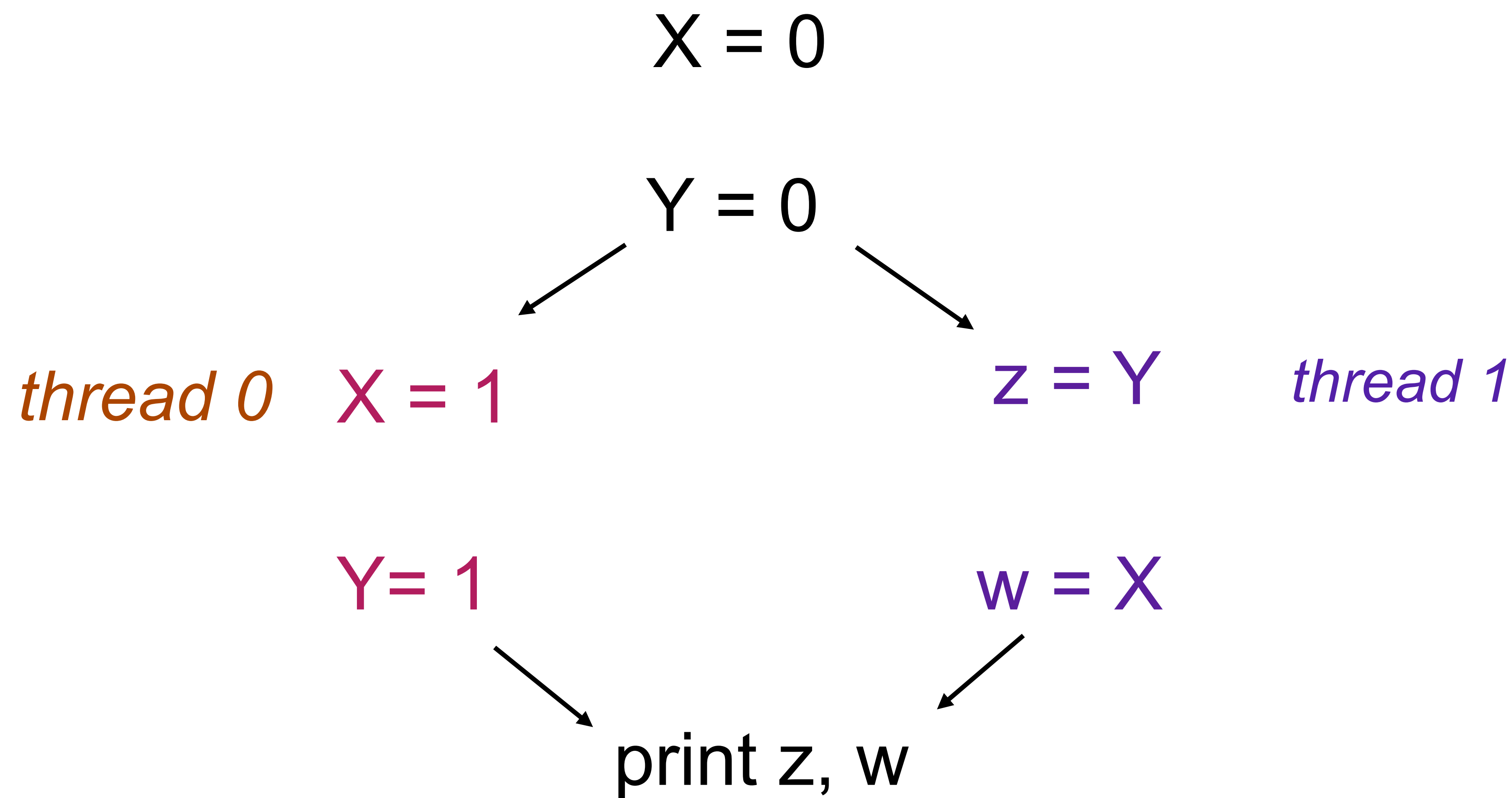
SC Example



Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: Not with sequential consistency

SC Example



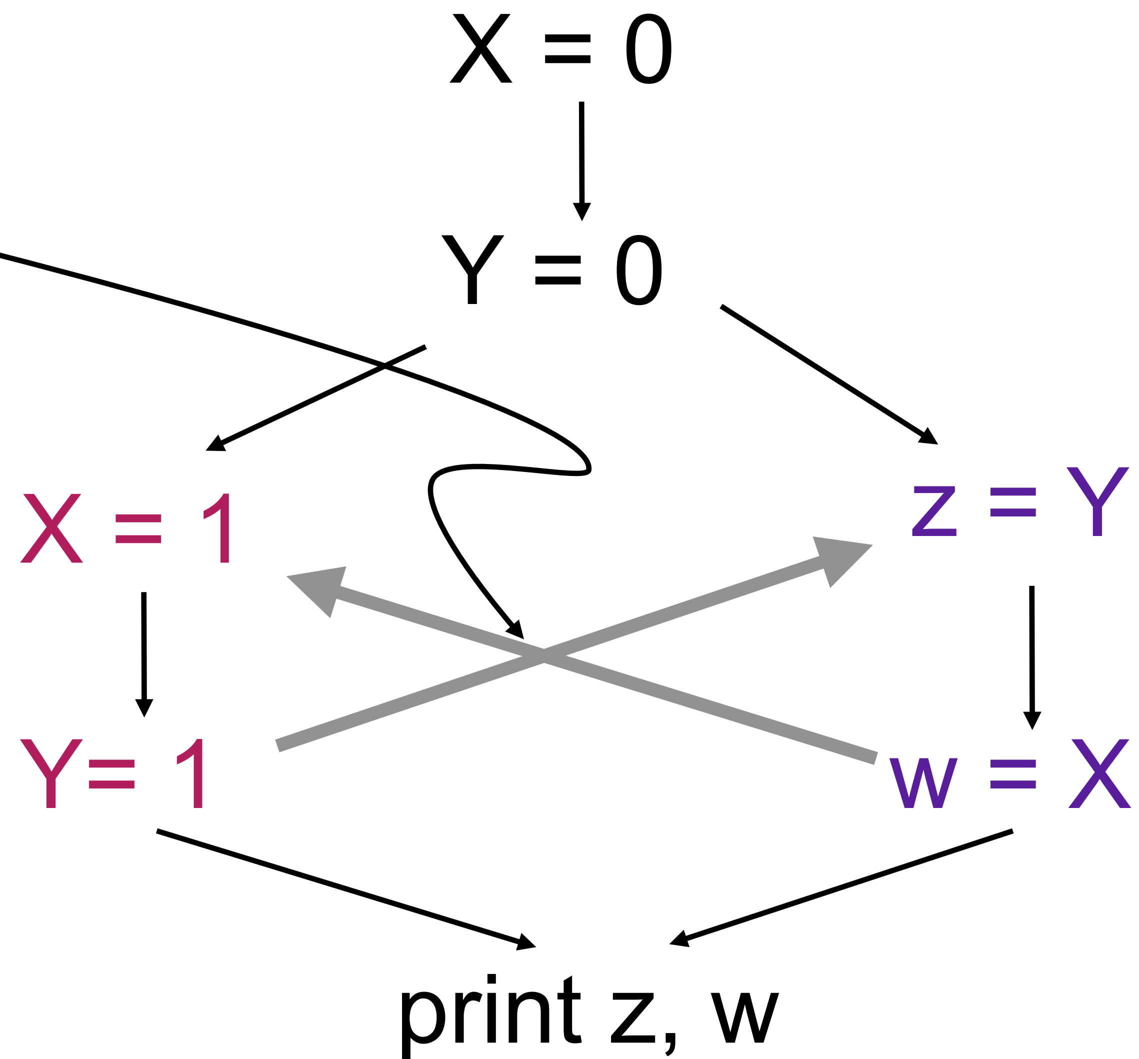
Question: Is it legal for ***z == 1*** and ***w == 0***?

For z == 1, "Y=1" must execute before "z=Y"

For w == 0, "w = X" must execute before "X=1"

Sequential Consistency

Relative execution order implied by assigned value



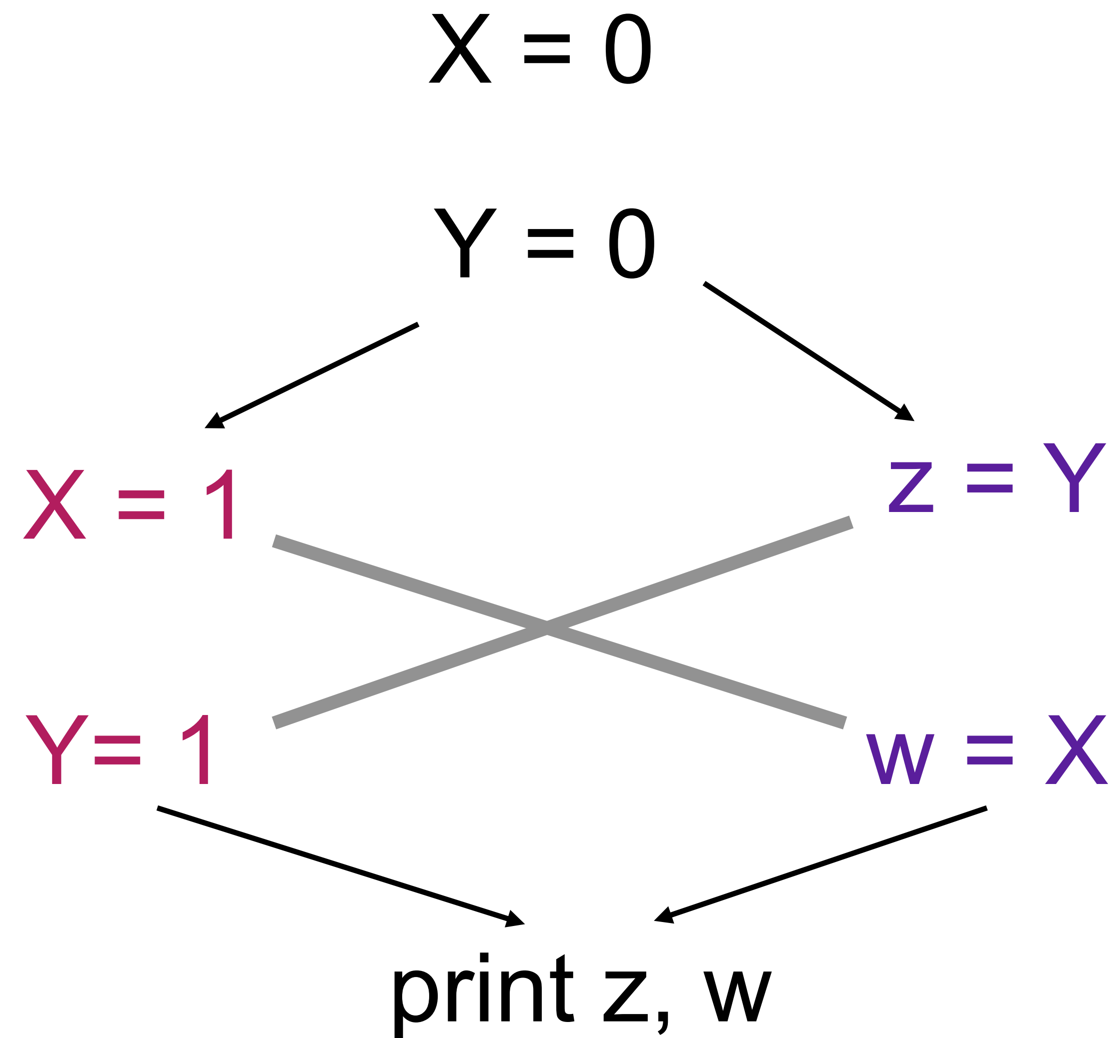
Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: ***NO.*** *For* ***z == 1*** and ***w == 0***, ordering in previous slide requires either "X=1" and "Y=1" to execute in a different order, or for "z=Y" or "w=X" to execute in a different order.

Many languages violate SC by default

Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: YES. Java semantics allow “X=1” and “Y=1” to execute in a different order, or for “z=Y” or “w=X” to execute in a different order.



Sequential Consistency (SC)

- Coherence says that a read will get the last value written for a variable
- Consistency is concerned with the interactions between writes to different variables, i.e., execution orders as seen in different threads are consistent with some definition of how orders should occur
- Sequential consistency (see Lamport paper) is when ... *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

We generally want programs to be SC

- After we parallelize the program the executions of the program should all give an answer such that ... *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
- Moreover, it is often good to have the program give the same answer as a sequential, one node, one core, one thread, etc. implementation of the algorithm
- It will be our responsibility as programmers to ensure this -- the hardware and software will not

We generally want programs to be SC

- It will be our responsibility as programmers to ensure this -- the hardware and software will not
- Hardware maintains coherence -- values read from a cache or memory will be the last value written
- Hardware typically maintains relaxed consistency -- within code running on a single thread, read orders with respect to writes *for a single variable* are maintained, write orders with respect to writes, *for a single variable*, are maintained.
- Instructions are provided to prevent re-orderings of other operations

Shared memory programming models

- Can either be a language, language extension, library or a combination
  - Java is a language and associated *virtual machine* that provides runtime support
  - OpenMP is a language extension (for C/C++ and Fortran) and an associated library (or *runtime*)
  - Pthreads (or *Posix Threads*) is a library with C/C++ and Fortran bindings

Back to our example of summing the elements of an array

- Same thing as in the bank example can happen
  - A thread gets a value of t,
  - gets interrupted (or maybe just holds its value in a register),
  - the other thread gets the same value of t, increments it, and then
  - the original thread increment its copy.

```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
  t += a[i];
}
t = t/n
```

The first update of t is lost.

Third (and correct but slow) try:

- use a *critical* section in the code
- executes the following (possible compound) statement atomically

```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
   t += a[i];
}
t = t/n
```

What is wrong with this?

Why this is slow

```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
   t = a[i];
}
t = t/n
```

i=1
t=a[0]

i=2
t=a[1]

i=3
t=a[2]

. . .

.

.

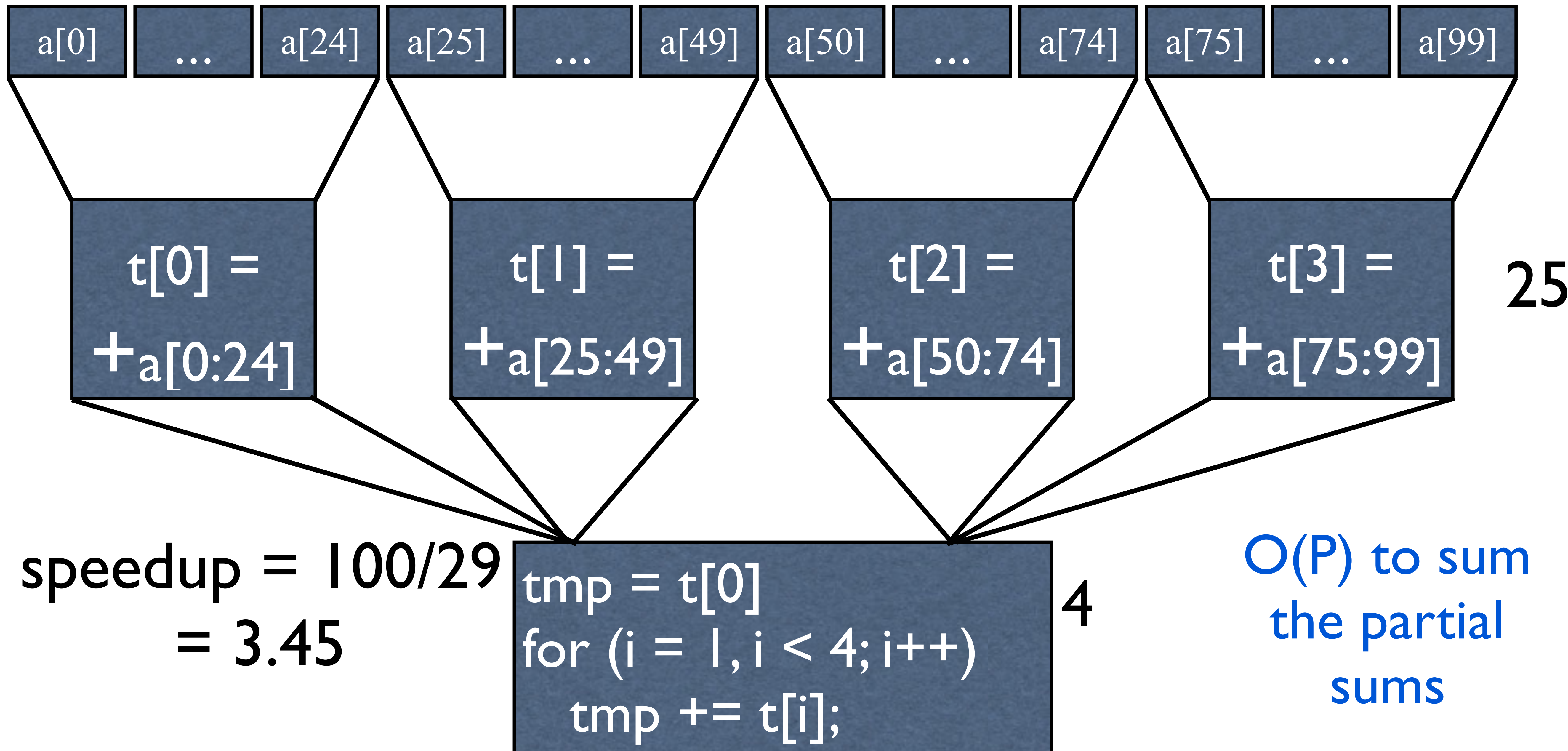
i=n-1
t=a[n-1]

time = O(n)

The operation we are trying to do is an example of a *reduction*

- Called a *reduction* because it takes something with d dimensions and reduces it to something with $d-k$, $k > 0$ dimensions
- Reductions on commutative operations can be done in parallel

A partially parallel reduction



How can we do this in OpenMP?

```
double t[4] = {0.0, 0.0, 0.0, 0.0}
int omp_set_num_threads(4);
#pragma omp parallel for
for (i=0; i < n; i++) {
   t[omp_get_thread_num( )] += a[i];
}
avg = 0;
for (i=0; i < 4; i++) }
   avg += t[i];
}
avg = avg / n;
```

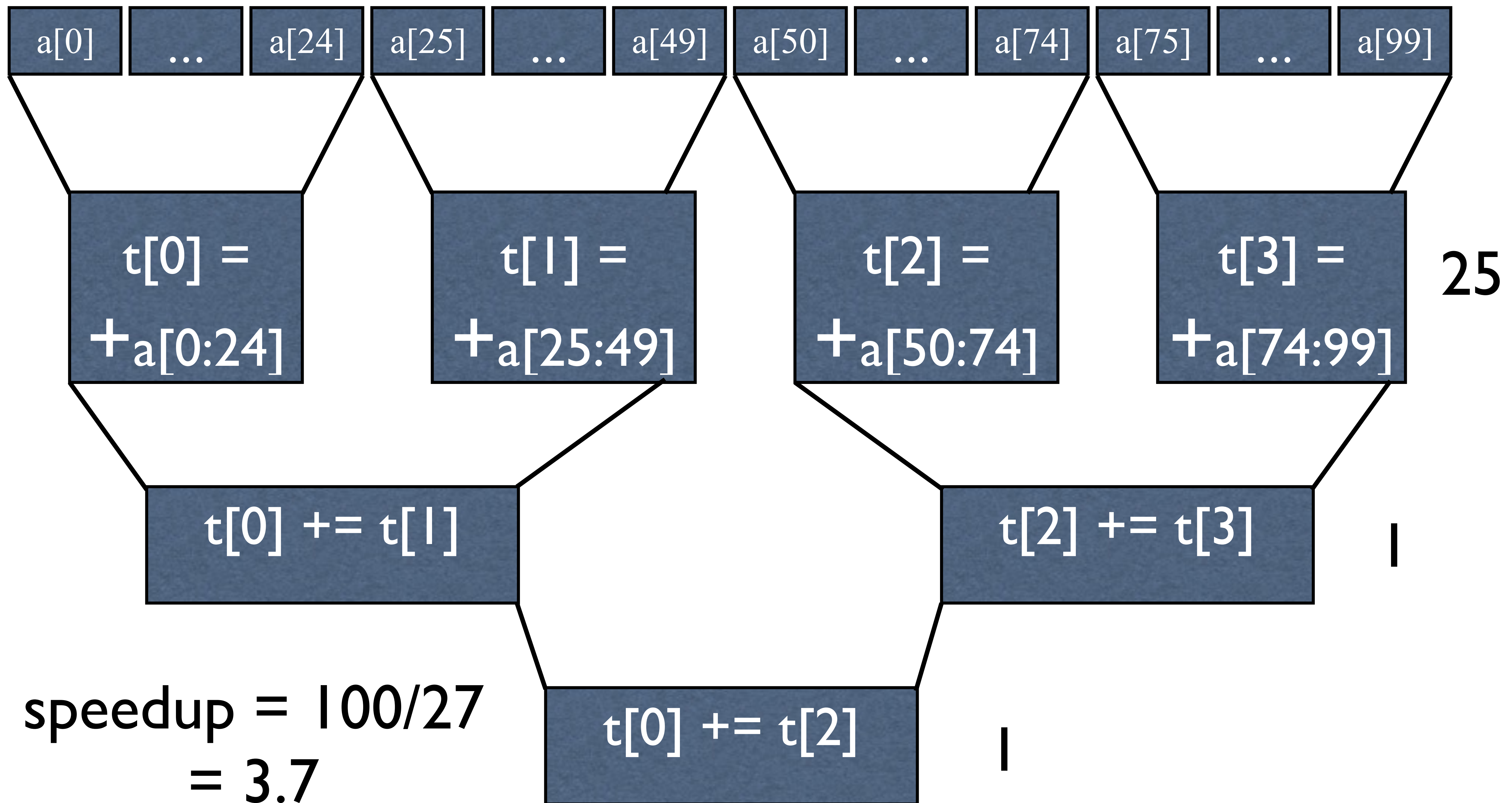
This is getting messy and we still are using a O(#threads) summation of the partial sums.

parallel

serial

OpenMP function

A better parallel reduction

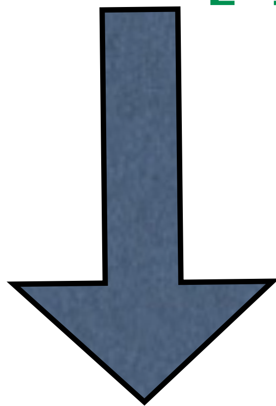


OpenMP provides an easy way to do this

- Reductions are common enough that OpenMP provides support for them
- reduction clause for omp parallel pragma
- specify variable and operation
- OpenMP takes care of creating temporaries, computing partial sums, and computing the final sum

Dot product example

```
t=0;
for (i=0; i < n; i++) {
  t = t + a[i]*c[i];
}
```



OpenMP makes *t* private, puts the partial sums for each thread into *t*, and then forms the full sum of *t* as shown earlier

```
t=0;
#pragma omp parallel for reduction(+:t)
for (i=0; i < n; i++) {
    t = t + (a[i]*c[i]);
}
```

Restrictions on Reductions

Operations on the reduction variable must be of the form

x = x op expr

x = expr op x (except subtraction)

x binop = expr

x++

++x

x--

--x

- ***x*** is a scalar variable in the list
- ***expr*** is a scalar expression that does not reference ***x***
- ***op*** is not overloaded, and is one of +, *, -, /, &, ^, |, &&, ||
- ***binop*** is not overloaded, and is one of +, *, -, /, &, ^, |

Why the restrictions on where *t* can appear?

```
#pragma omp parallel for reduction(+:t)
// each element of a[i] = 1
for (i=0; i<n; i++) {
    b[i] = t;
     t += a[i];
}
```

- In the sequential loop, at the end of iteration i, $t = i+1$.
- Let s_t be the starting iteration for the thread t, then
  $$s_t = (tid-1)*ceil(n/\#threads)+i+1$$
- If executed as a recurrence using a static distribution of iterations, at the end of iteration i, $t=i-s_t$,
- Thus, if $n = 100$, thread 3 executes iterations $50...74$, and in iteration 60, $i = 11$
- This means $b[61] = 11$, not 61
- making it work right is, in general, hard to do efficiently.
- Thus the OpenMP restriction on where t can appear

Improving performance of parallel loops

```
#pragma omp parallel for reduction(+:t)
for (i=0; i < n; i++) {
    t = t + (a[i]*c[i]);
}
```

- Parallel loop startup and teardown has a cost
- Parallel loops with few iterations can lead to slowdowns -- if clause allows us to avoid this
- This overhead is one reason to try and parallelize outermost loops.

```
#pragma omp parallel for reduction(+:t) if (n>1000)
for (i=0; i < n; i++) {
    t = t + (a[i]*c[i]);
}
```

Assigning iterations to threads (thread scheduling)

- The schedule clause can guide how iterations of a loop are assigned to threads
- Two kinds of schedules:
  - static: iterations are assigned to threads at the start of the loop. Low overhead but possible load balance issues.
  - dynamic: some iterations are assigned at the start of the loop, others as the loop progresses. Higher overheads but better load balance.
- A *chunk* is a contiguous set of iterations

The schedule clause - static

- schedule(*type[, chunk])* where "*[ ]*" indicates optional
- *(type [,chunk])* is
  - (static): chunks of ~ *n/t* iterations per thread, no chunk specified. The default.
  - (static, chunk): chunks of size *chunk* distributed round-robin. No *chunk* specified means *chunk = 1*

The schedule clause - dynamic

- schedule(*type[, chunk])* where "*[ ]*" indicates optional
- *(type [,chunk])* is
  - (dynamic): chunks of size of *1* iteration distributed dynamically
  - (dynamic, chunk): chunks of size *chunk* distributed dynamically

Static

	thread 0	thread 1	thread 2
Chunk = 1	0, 3, 6, 9, 12	1, 4, 7, 10, 13	2, 5, 8, 11, 14

	thread 0	thread 1	thread 2
Chunk = 2	0, 1, 6, 7, 12, 13	2, 3, 8, 9, 14, 15	4, 5, 10, 11, 16, 17

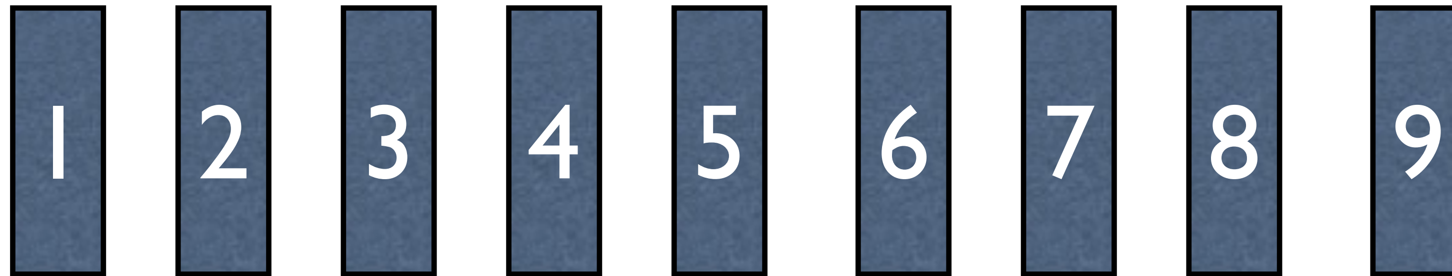
With no chunk size specified, the iterations are divided as evenly as possible among processors, with one chunk per processor

With *dynamic* chunks go to processors as work needed.

The schedule clause

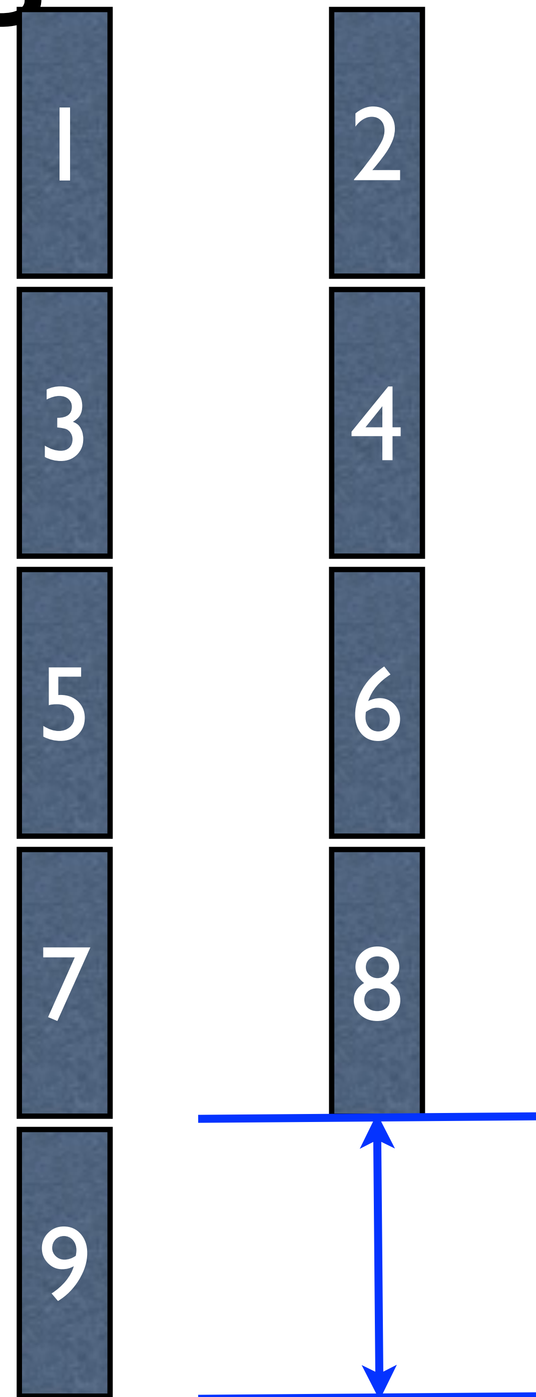
- schedule(*type[, chunk])* (type [,chunk]) is
  - (guided,chunk): uses *guided self scheduling* heuristic. Starts with big chunks and decreases to a minimum chunk size of *chunk*
  - runtime - type depends on value of OMP_SCHEDULE environment variable, e.g. setenv OMP_SCHEDULE=”static,1”

Guided with two threads example



Dynamic schedule with large blocks

Large blocks reduce scheduling costs, but lead to large load imbalance

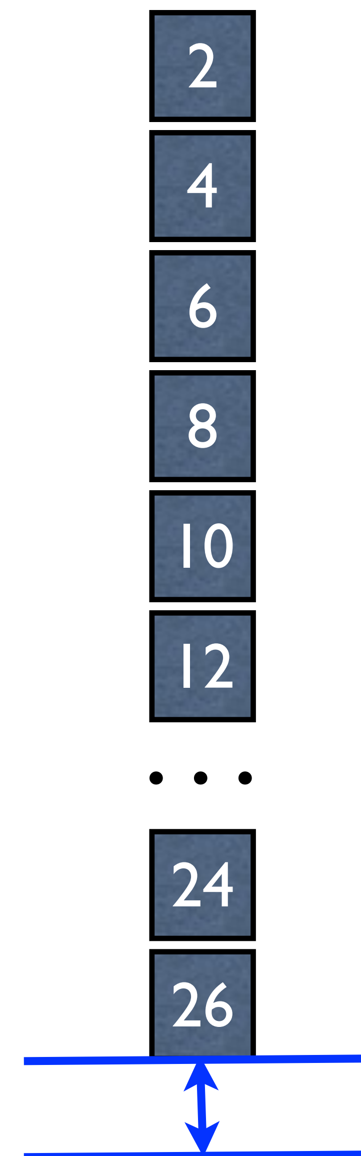


Dynamic schedule with small blocks

Thread 0 Thread 1

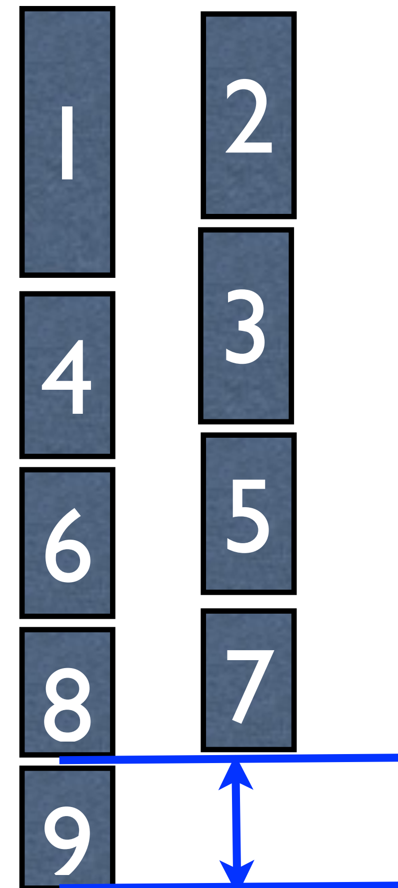
Small blocks have a smaller load imbalance, but with higher scheduling costs.

Would like the best of both methods.



Guided with two threads

By starting out with larger blocks, and then ending with small ones, scheduling overhead and load imbalance can both be minimized.



The nowait clause

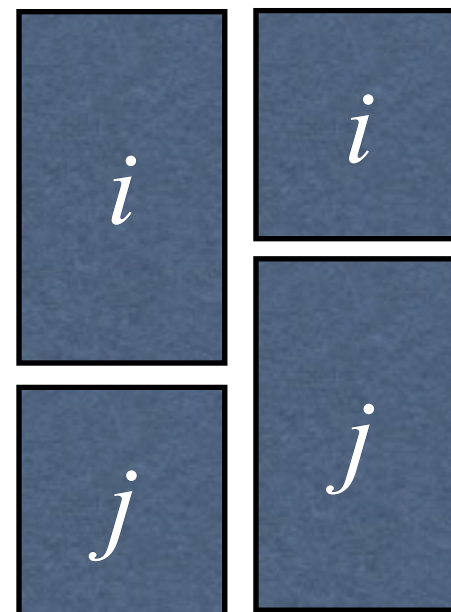
```
#pragma omp parallel for
for (i=0; i < n; i++) {
  if (a[i] > 0) a[i] += b[i];
}
```

barrier here by default

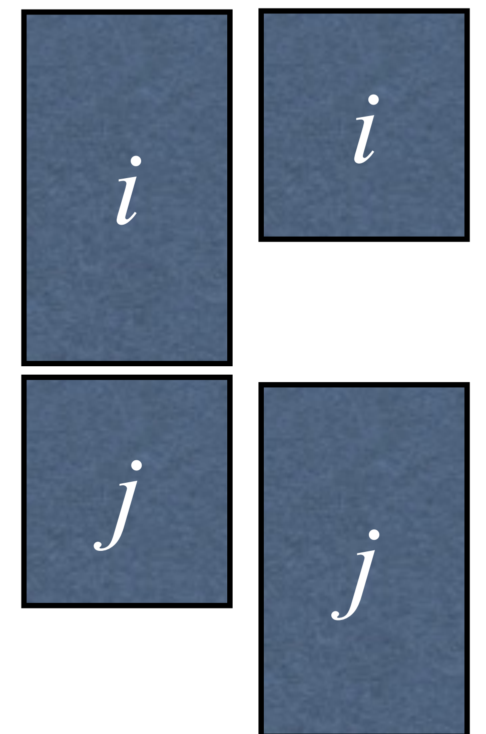
```
#pragma omp parallel for nowait
for (i=0; i < n; i++) {
  if (a[i] < 0) a[i] -= b[i];
}
```

time

with nowait



without nowait

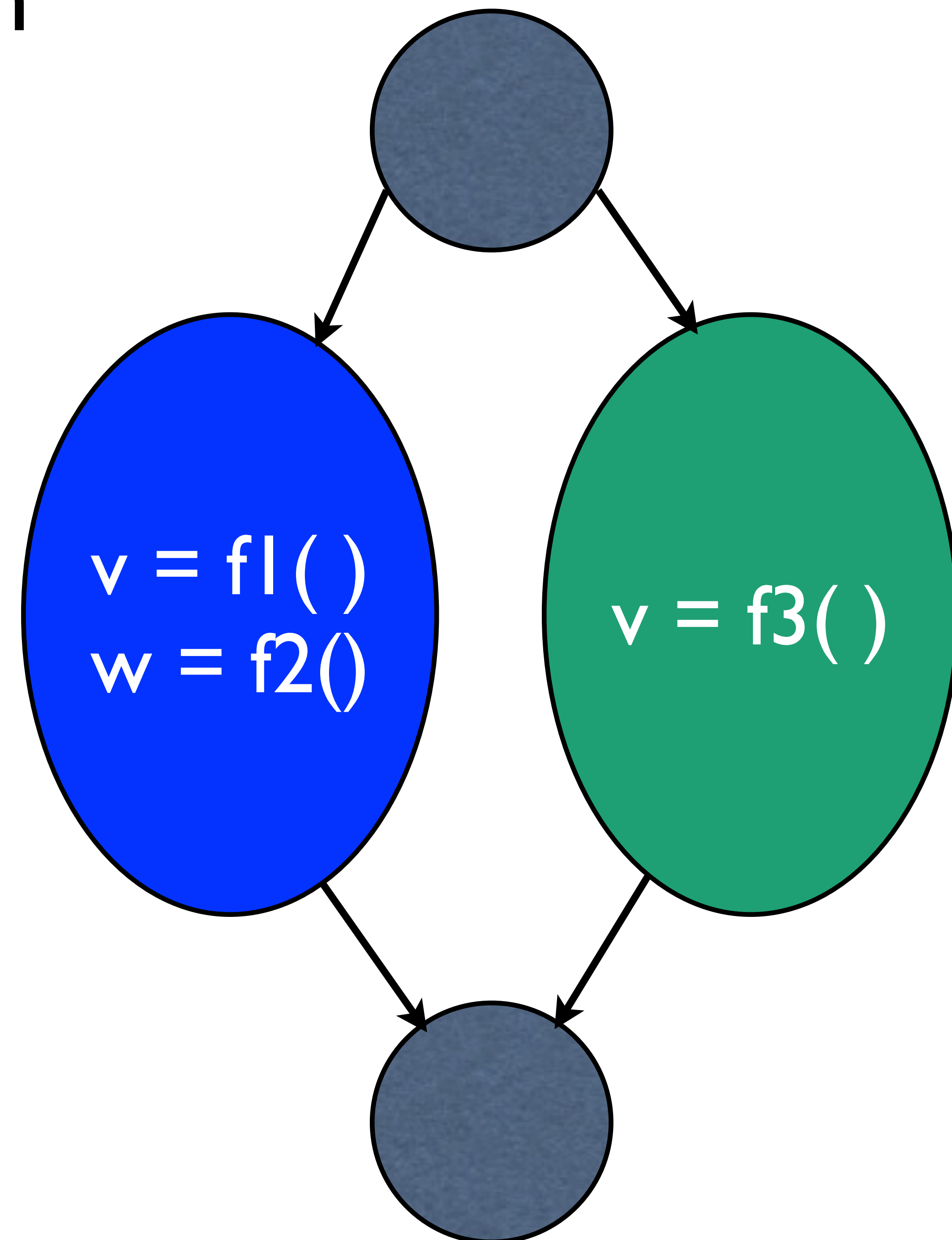


Only the static distribution with the same bounds guarantees the same thread will execute the same iterations from both loops.

The sections pragma

Used to specify *task* parallelism

```
#pragma omp parallel sections
   {
#pragma omp section /* optional */
   {
   v = f1( )
   w = f2( )
   }
#pragma omp section
  v = f3( )
}
```



The parallel pragma

```
#pragma omp parallel private(w)
{
  w = getWork (q);
  while (w != NULL) {
    doWork(w);
    w = getWork(q);
  }
}
```

- every processor executes the statement following the *parallel* pragma
- Parallelism of useful work in the example because independent and different work pulled of of q
- q needs to be thread safe

The parallel pragma

```
#pragma omp parallel private(w)
{
#pragma omp critical
  w = getWork (q);
  while (w != NULL) {
    doWork(w);
#pragma omp critical
    w = getWork(q);
  }
}
```

- If data structure pointed to by *q* is not thread safe, need to synchronize it in your code
- One way is to use a *critical* clause

single and *master* clauses exist.

The single directive

```
#pragma omp parallel private(w)
{
  w = getWork (q);
  while (w != NULL) {
    doWork(w);
    w = getWork(q);
  }
  #pragma omp single
    fprintf(“finishing work”);
}
```

Requires statement following the pragma to be executed by the master thread.

Differs from critical in that critical lets the statement execute on many threads, just one at a time.

The master directive

```
#pragma omp parallel private(w)
{
  w = getWork (q);
  while (w != NULL) {
    doWork(w);
    w = getWork(q);
  }
  #pragma omp master
    fprintf("finishing work");
}
```

Requires statement following the pragma to be executed by the master thread.

Often the master thread is thread 0, but this is implementation dependent. Master thread is the same thread for the life of the program.

Cannot use single/ master with *for*

```
#pragma omp parallel for
for (i=0; i < n; i++) {
   if (a[i] > 0) {
      a[i] += b[i];
#pragma omp single
      printf("exiting");
   }
}
```

Many different instances of the single

Does OpenMP provide a way to specify:

- what parts of the program execute in parallel with one another
- how the work is distributed across different cores
- the order that reads and writes to memory will take place
- that a sequence of accesses to a variable will occur *atomically* or without interference from other threads.
- **And,** ideally, it will do this while giving *good performance* and allowing *maintainable programs* to be written.

What executes in parallel?

```
c = 57.0;
for (i=0; i < n; i++) {
  a[i] = c + a[i]*b[i]
}
```

```
c = 57.0
#pragma omp parallel for
for (i=0; i < n; i++) {
  a[i] = + c + a[i]*b[i]
}
```

- *pragma* appears like a comment to a non-OpenMP compiler
- pragma requests parallel code to be produced for the following for loop

The order that reads and writes to memory occur

```
c = 57.0
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c + a[i]*b[i]
}
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c + a[i]*b[i]
}
```

- Within an iteration, access to data appears in-order
- Across iterations, no order is implied. *Races* lead to undefined programs
- Across loops, an implicit *barrier* prevents a loop from starting execution until all iterations and writes (stores) to memory in the previous loop are finished
- Parallel constructs execute after preceding sequential constructs finish

Relaxing the order that reads and writes to memory occur

```
c = 57.0
#pragma omp parallel for schedule(static) nowait
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}

#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

no barrier

The *nowait* clause allows a thread to begin executing its part of the code after the nowait loop as soon as it finishes its part of the nowait loop

Accessing variables without interference from other threads

```
#pragma omp parallel for
for (i=0; i < n; i++) {
   a = a + b[i]
}
```

Dangerous -- all iterations are updating *a* at the same time -- a *race* (or *data race*).

```
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
   a = a + b[i];
}
```

Inefficient but correct -- *critical* pragma allows only one thread to execute the next statement at a time. Potentially slow -- *but ok if you have enough work in the rest of the loop to make it worthwhile.*

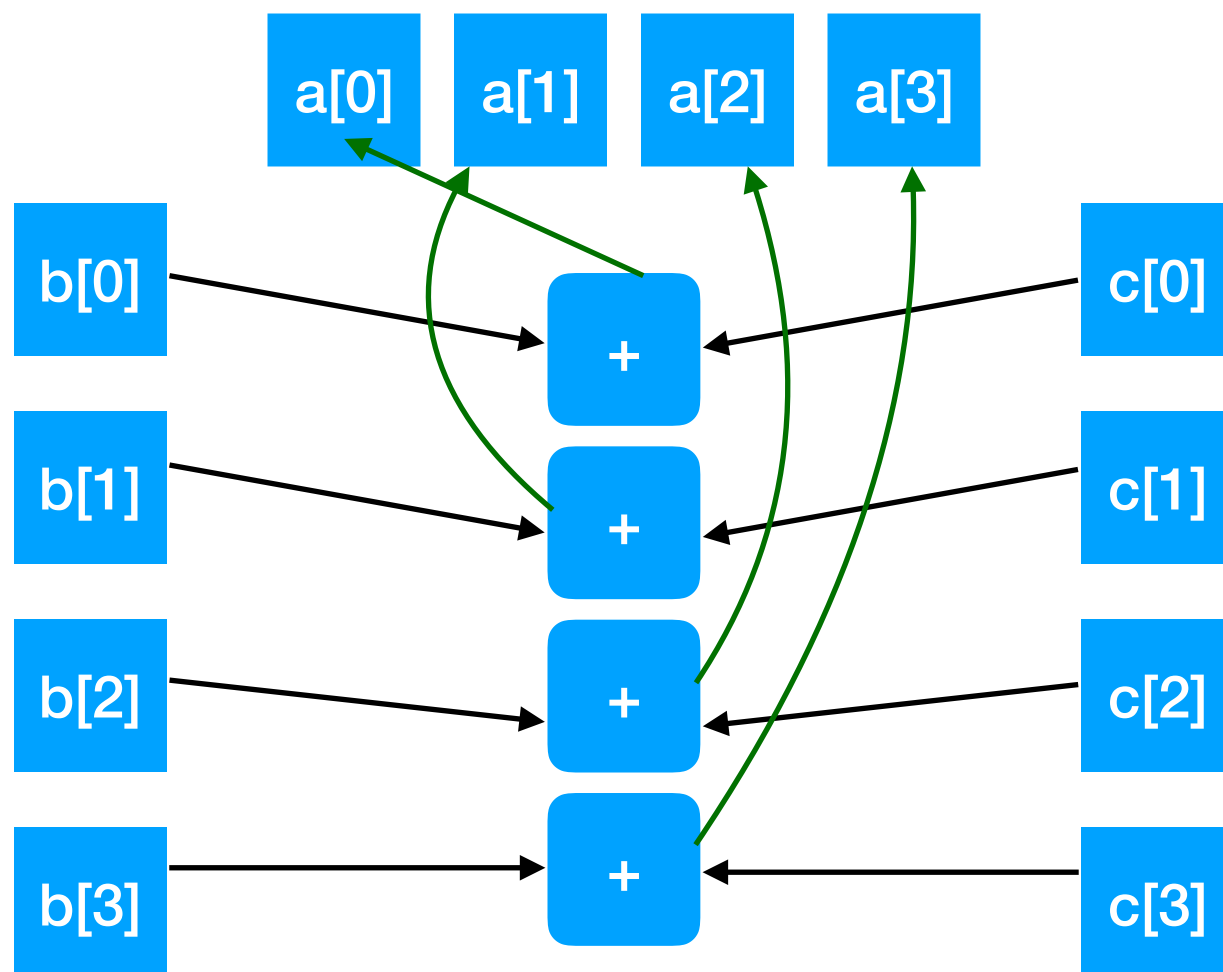
Other kinds of parallelism

Vector parallelism (SIMD parallelization)

Why vectors?

```
For (int i = 0; i < n; i++) {
   a[i] = b[i]*c[i];
}
```

```
For (int i = 0; i < n; i+=4) {
   a[i] = b[i]*c[i];
   a[i+1] = b[i+1]*c[i+1];
   a[i] = b[i+2]*c[i+2];
   a[i] = b[i+2]*c[i+2];
}
```

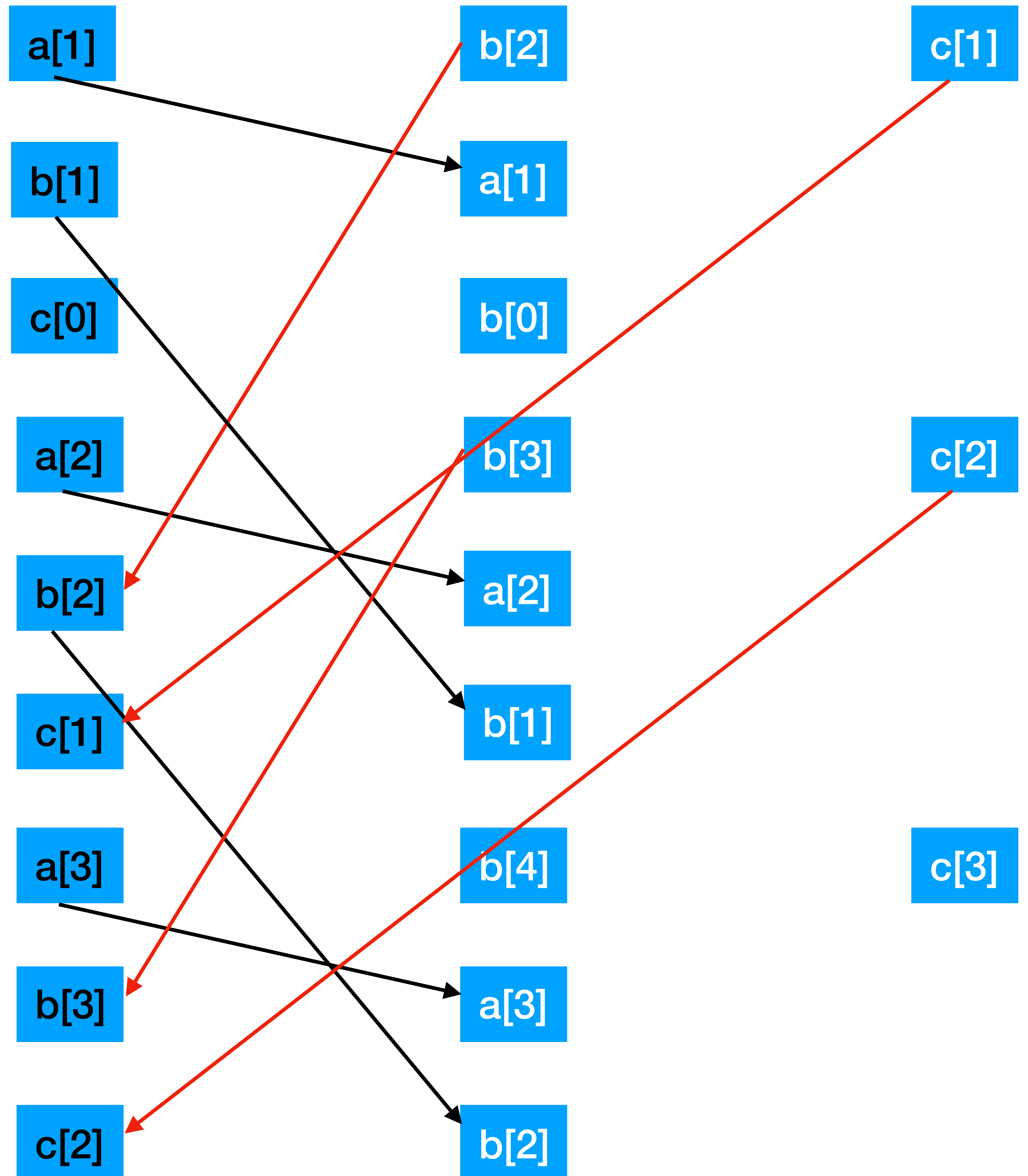


- Normal multi-functional unit processors need circuitry to control and fire the functional units
- This is not under programmer control — the programmer only specifies the instructions to be executed, not the functional unit that executes the instruction.
- Hardware must detect availability of operands and functional unit, and schedule the operation onto a particular hardware functional unit. Enables *out-of-order* execution.
    - See “scoreboard” and “Tomasulo algorithm” for details
    - Robert Tomasulo, IBM, 1967, IBM 360, Model 91
    - Derivatives of it are used in most modern architectures

Dataflow takes this one step further

```
for (int i = 0; i < n; i++) {
   a[i] = b[i+1] + c[i]
   b[i] = a[i-1]
   c[i-1] = b[i-1]
}
```

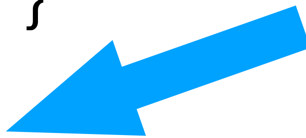
```
for (int i = 1; i < n/3; i++) {
   a[i] = b[i+1] + c[i]
   b[i] = a[i-1]
   c[i-1] = b[i-1]
   a[i+1] = b[i+2] + c[i+1]
   b[i] = a[i]
   c[i] = b[i]
   a[i+2] = b[i+3] + c[i+2]
   b[i] = a[i+1]
   c[i+1] = b[i+1]

}
```



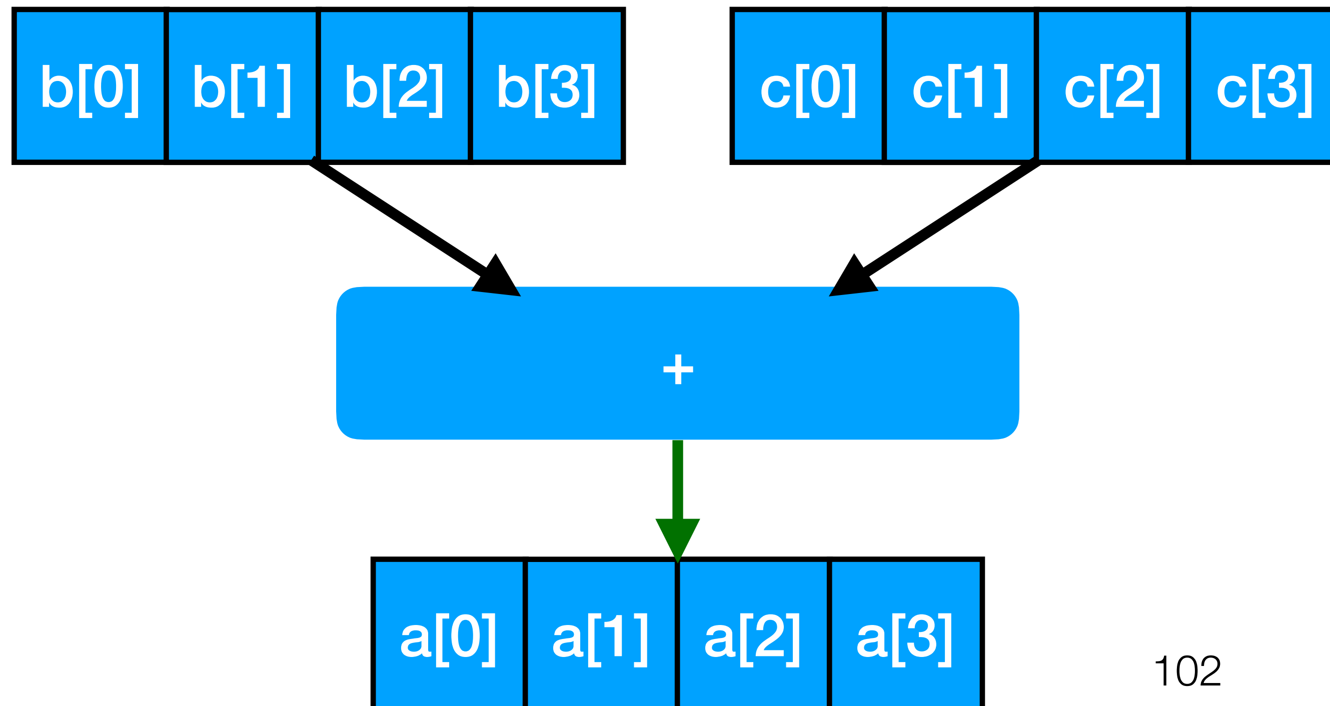
Why vectors?

```
For (int i = 0; i < n; i++) {
   a[i] = b[i]*c[i];
}
```

```
For (int i = 0; i < n; i+=4) {
   a[i] = b[i]*c[i];
   a[i+1] = b[i+1]*c[i+1];
   a[i] = b[i+2]*c[i+2];
   a[i] = b[i+2]*c[i+2];
}
```



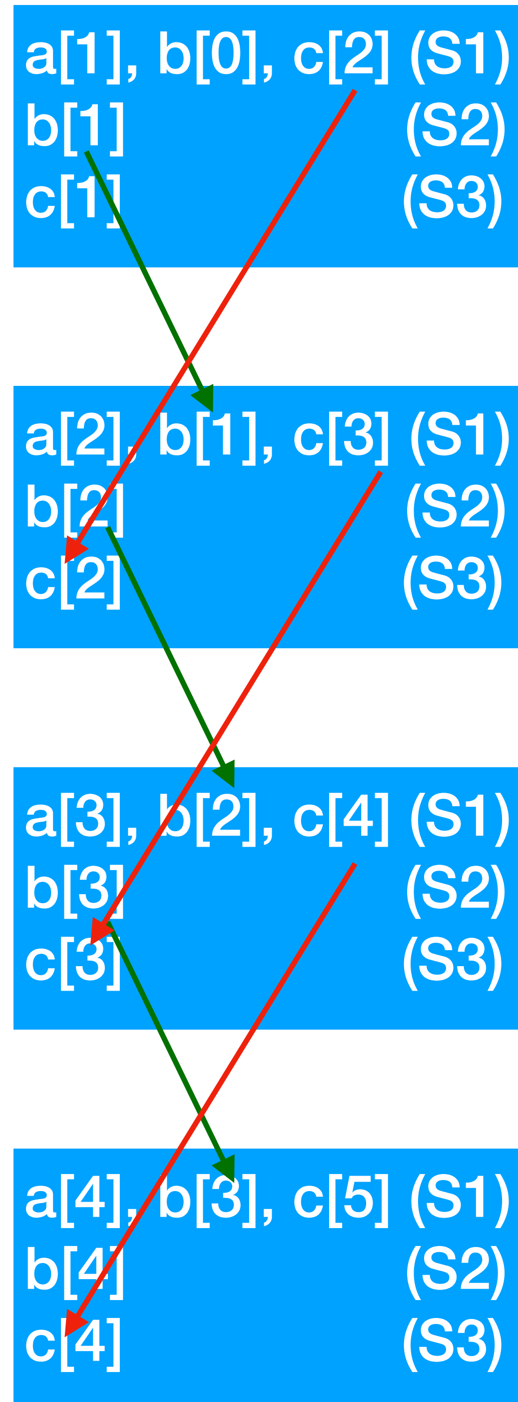
```
For (int i = 0; i < n; i+=4) {
   ldv rv1, b[i]
   ldv rv2, c[i]
   vadd rv3, rv1, rv2
}
```



- With vector units there are architected vector registers and vector functional units
- These work on groups, or vectors of operands and operations
- Programmer/compiler generates the instructions
- Control in hardware is almost no more complicated than a scalar functional unit
- Allows more operations to be done per clock with small increase in processor complexity

Vector parallelization

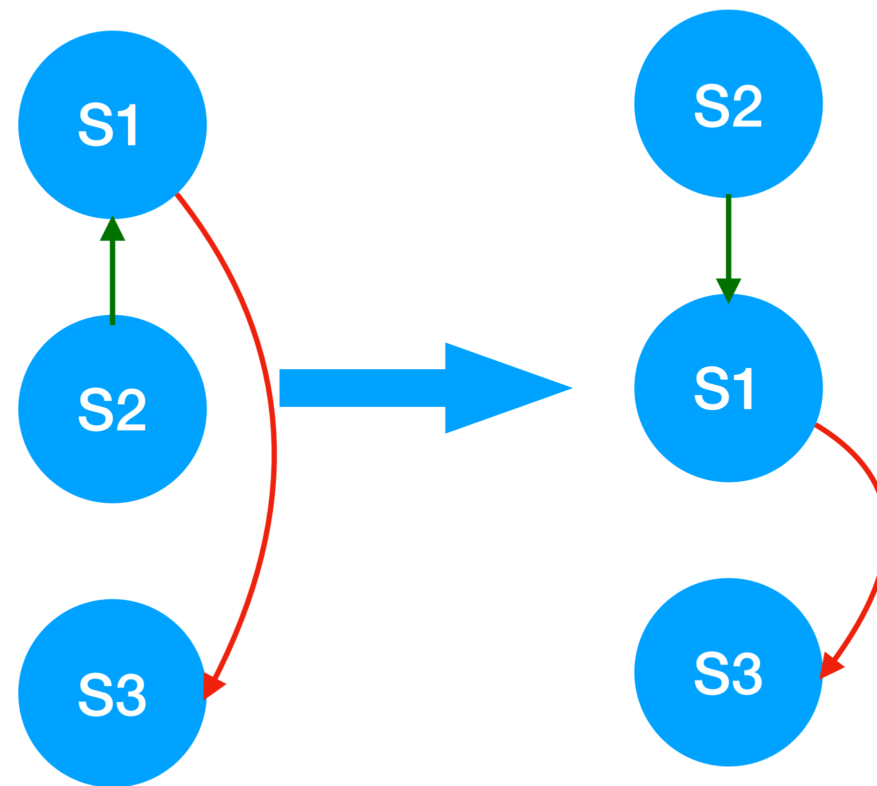
```
for (int i = 1, i < n, i++) {

   a[i] = b[i-1] + c[i+1]; (S1)

   b[i] = d[i] + e[i]; (S2)

   c[i] = f[i] + g[i]; (S3)

}
```



Dependences go from earlier to later statements. This is not good, as executing 4 iterations of S1 before S2 will cause S1 to get stale values.

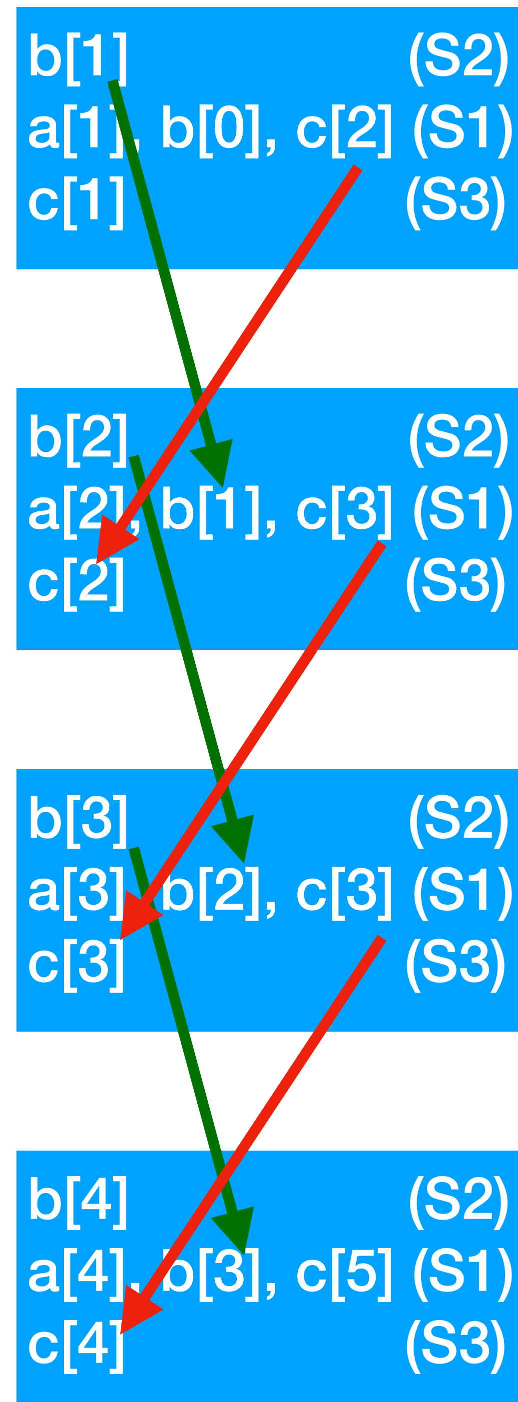
Vector parallelization

```
for (int i = 1, i < n, i++) {

   a[i] = b[i-1] + c[i+1]; (S1)

   b[i] = d[i] + e[i]; (S2)

   c[i] = f[i] + g[i]; (S3)

}
```



Vector parallelization

```
for (int i = 1, i < n, i++) {

   b[i] = d[i] + e[i]; (S2)

   a[i] = b[i-1] + c[i+1]; (S1)

   c[i] = f[i] + g[i]; (S3)

}
```

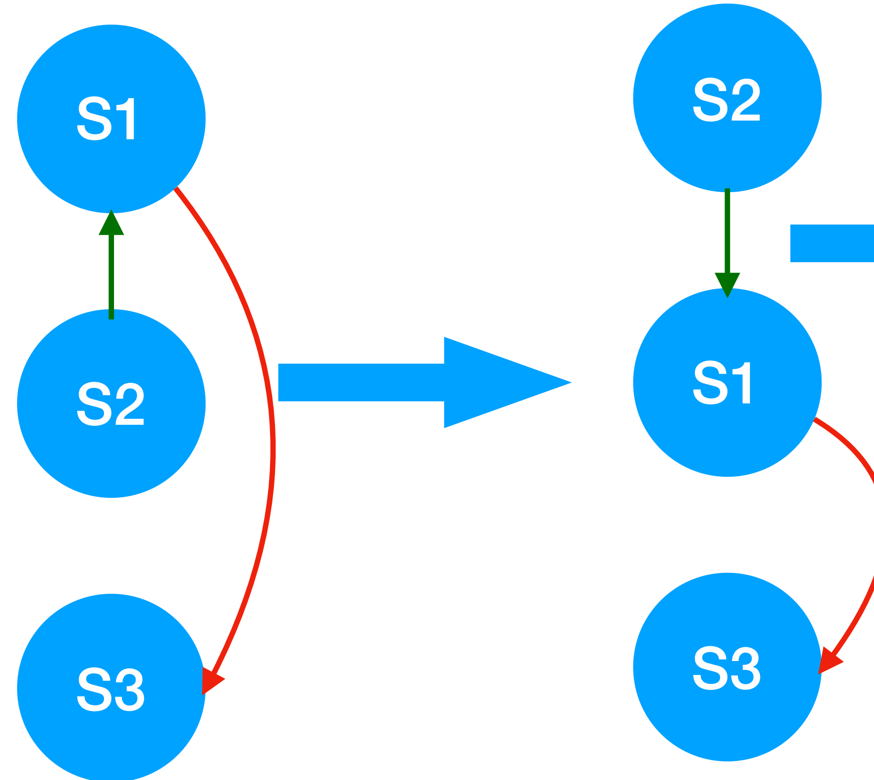


Dependences go from earlier statements to later statements.

Now we can execute all S2 before all S1, and all S1 before all S3, and no dependences will be violated.

Vector parallelization

```
for (int i = 1, i < n, i++) {

   a[i] = b[i-1] + c[i+1]; (S1)

   b[i] = d[i] + e[i]; (S2)

   c[i] = f[i] + g[i]; (S3)

}
```




```
for (int i = 1, i < n, i++) {

   b[i] = d[i] + e[i]; (S2)

   a[i] = b[i-1] + c[i+1]; (S1)

   c[i] = f[i] + g[i]; (S3)

}
```

```
for (int i = 1, i < n, i++) {
   b[i] = d[i] + e[i]; (S2)
}

for (int i = 1, i < n, i++) {
   a[i] = b[i-1] + c[i+1]; (S1)
}

for (int i = 1, i < n, i++) {
   c[i] = f[i] + g[i]; (S3)
}
```

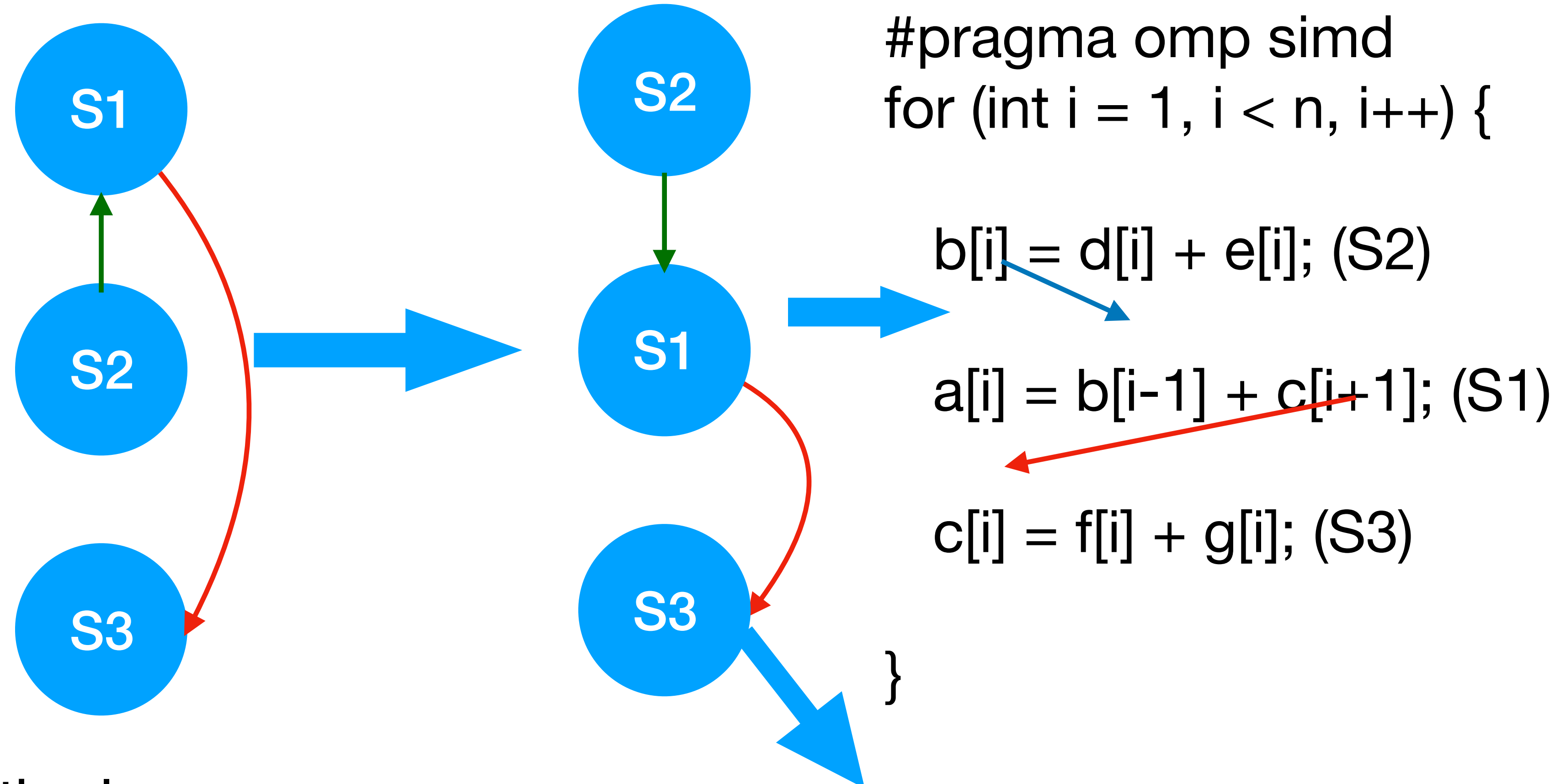
```
for (int i = 1, i < n, i+=4) {
   vadd b[i], d[i], e[i]; (S2)
}

for (int i = 1, i < n, i+=4) {
   vadd a[i], b[i-1], c[i+1]; (S1)
}

for (int i = 1, i < n, i+=4) {
   vadd c[i], f[i], g[i]; (S3)
}
```

Modern server-grade high performance processors can do 32 or more vector operations at a time.

GPUs, as we will see, can do thousands of operations at a time

Vector parallelization

```
for (int i = 1, i < n, i++) {

  a[i] = b[i-1] + c[i+1]; (S1)

  b[i] = d[i] + e[i]; (S2)

  c[i] = f[i] + g[i]; (S3)

}
```



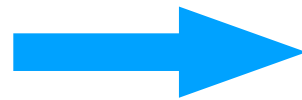
`#pragma omp simd` runs the loop on a single core using the vector units in that core.

`#pragma omp parallel for simd` runs the loop on the available cores, using the vector units in that core.

```
#pragma omp parallel for simd
for (int i = 1, i < n, i++) {

  b[i] = d[i] + e[i]; (S2)

  a[i] = b[i-1] + c[i+1]; (S1)

  c[i] = f[i] + g[i]; (S3)

}
```

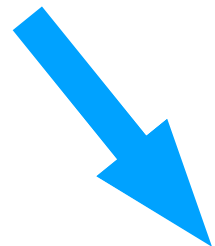
Vector parallelization with OMP

```
int max(int b, int c) {
   if (b > c) return b;
   else return c;
}
…
for (int i = 1, i < n, i++) {
   a[i] = max(b[i],c[i]);
}
```



```
#pragma omp declare simd
int max(int b, int c) {
   if (b > c) return b;
   else return c;
}
…
#pragma omp parallel for simd
for (int i = 1, i < n, i++) {
   a[i] = max(b[i], c[i];
}
```

Executes using vector units of a multiple cores.



```
#pragma omp declare simd
int max(int b, int c) {
   if (b > c) return b;
   else return c;
}
…
#pragma omp simd
for (int i = 1, i < n, i++) {
   a[i] = max(b[i], c[i];
}
```

Executes using vector units of a multiple cores.