Basic OpenMP

Last updated 12:38, January 14. Previously updated January 11, 2019 at 3:08PM

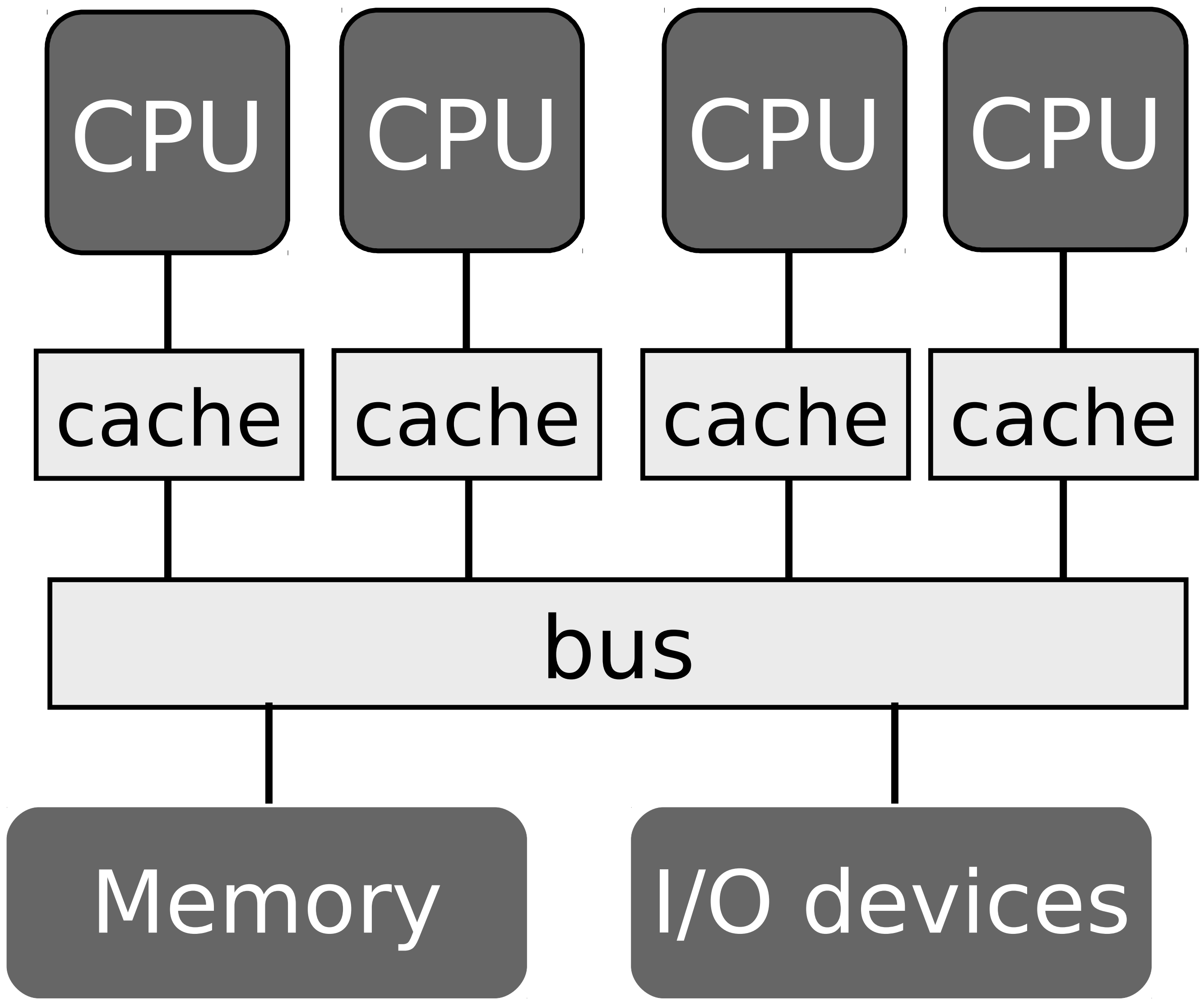
You should now have a scholar account

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

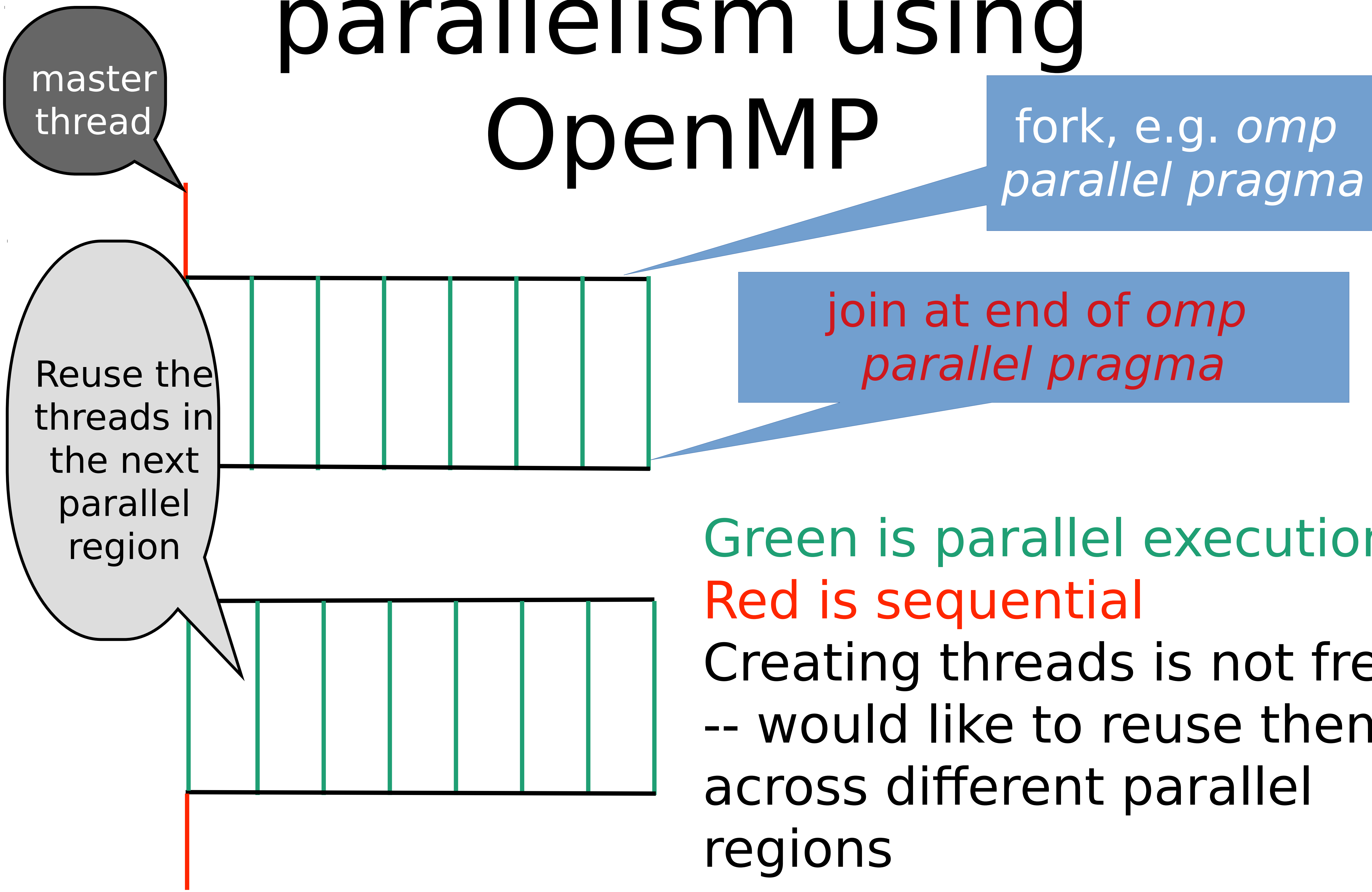
Uniform memory access shared memory machine is assumed



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

Typical thread level parallelism using OpenMP



Green is parallel execution

Red is sequential

Creating threads is not free -- would like to reuse them across different parallel regions

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;
```

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 *when the loop begins executing*

Each thread has an execution context

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

shared/private

Example of context

Consider the program below:

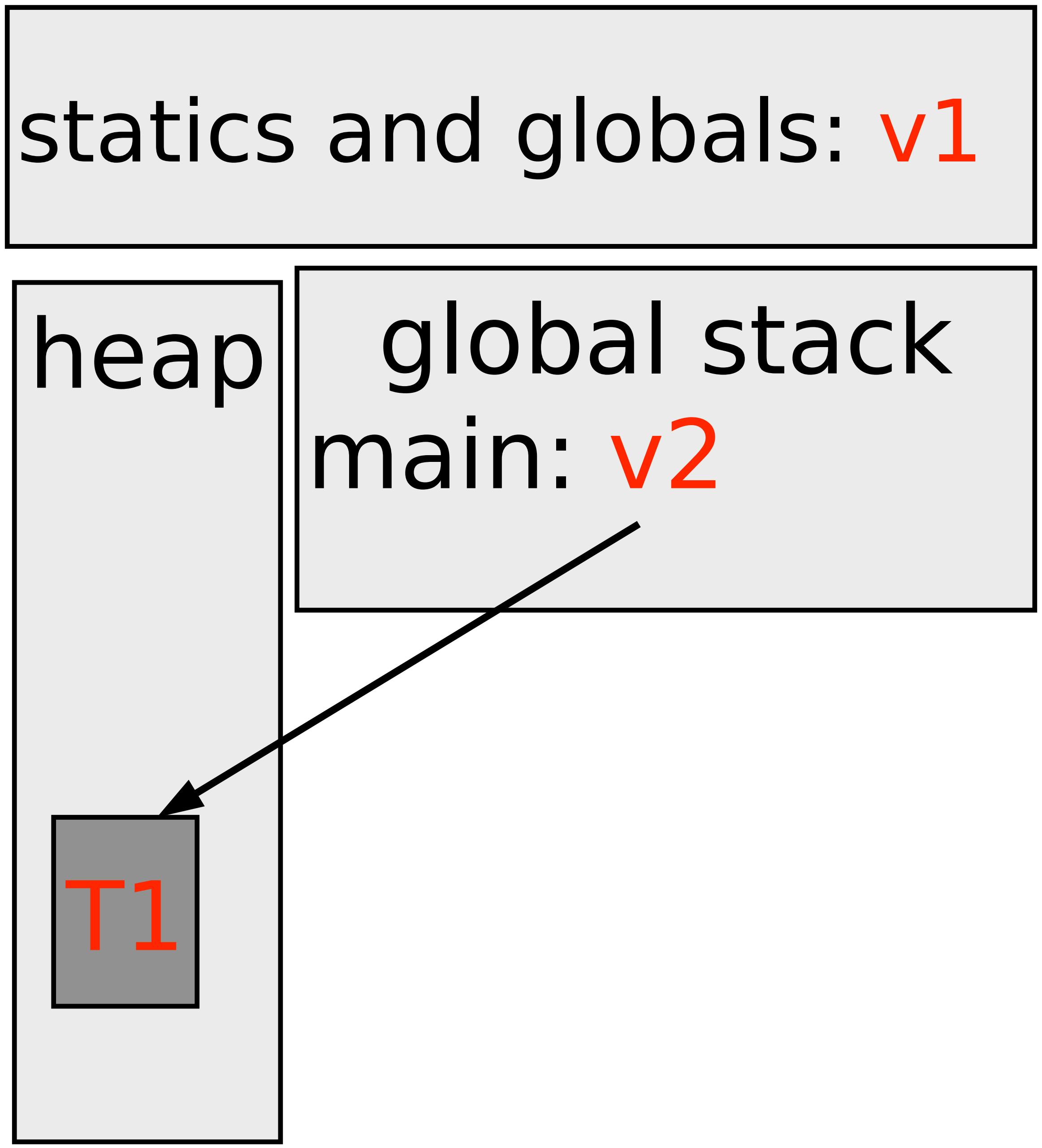
```
int v1;
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for
   for (int i=0; i < n; i++) {
      int v4;
      T1 *v5 = malloc(sizeof(T1));
}}
```

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

Context before call to f1

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

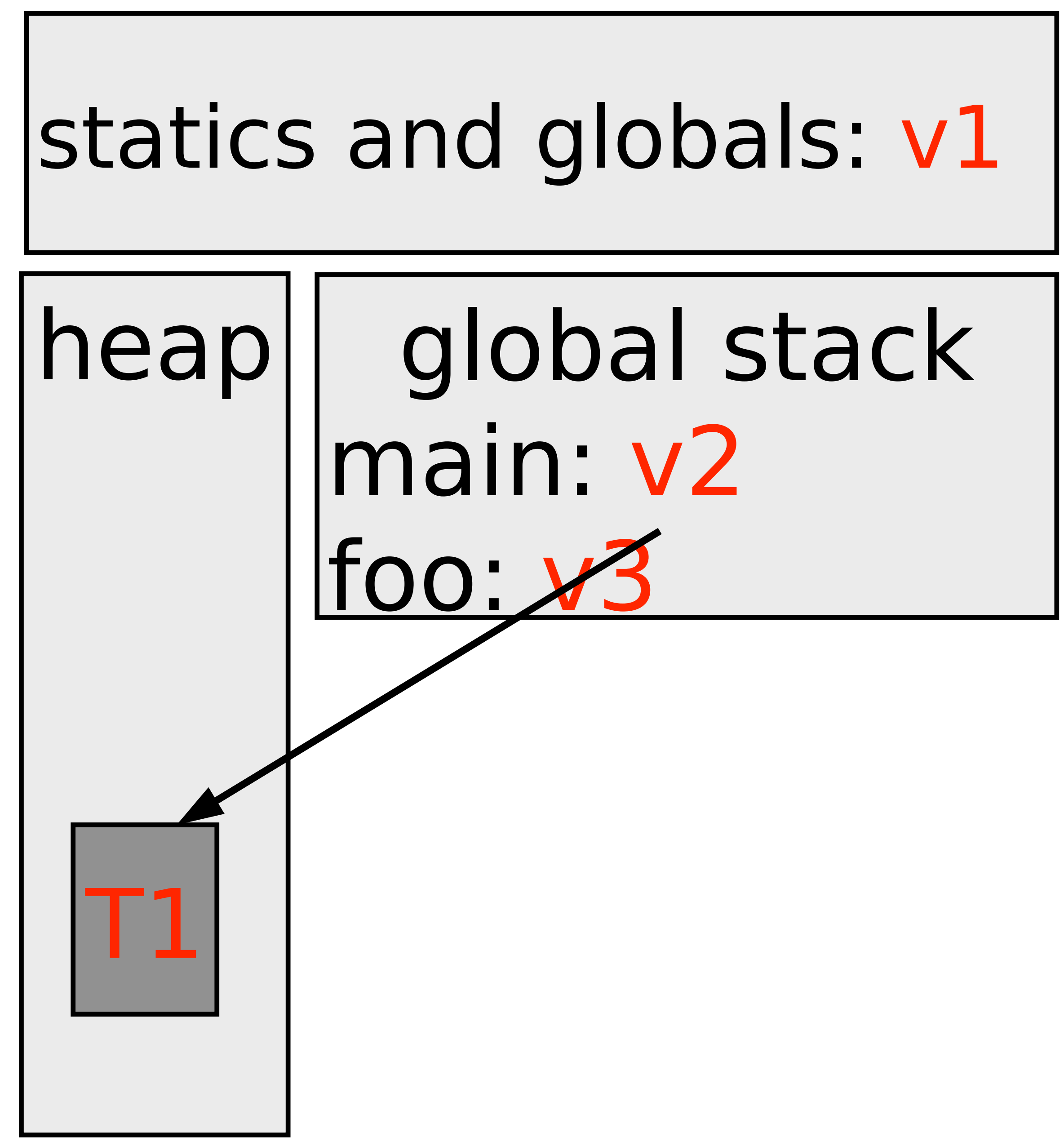
```
int v1;
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for
   for (int i=0; i < n; i++) {
      int v4;
      T1 *v5 = malloc(sizeof(T1));
}}
```



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

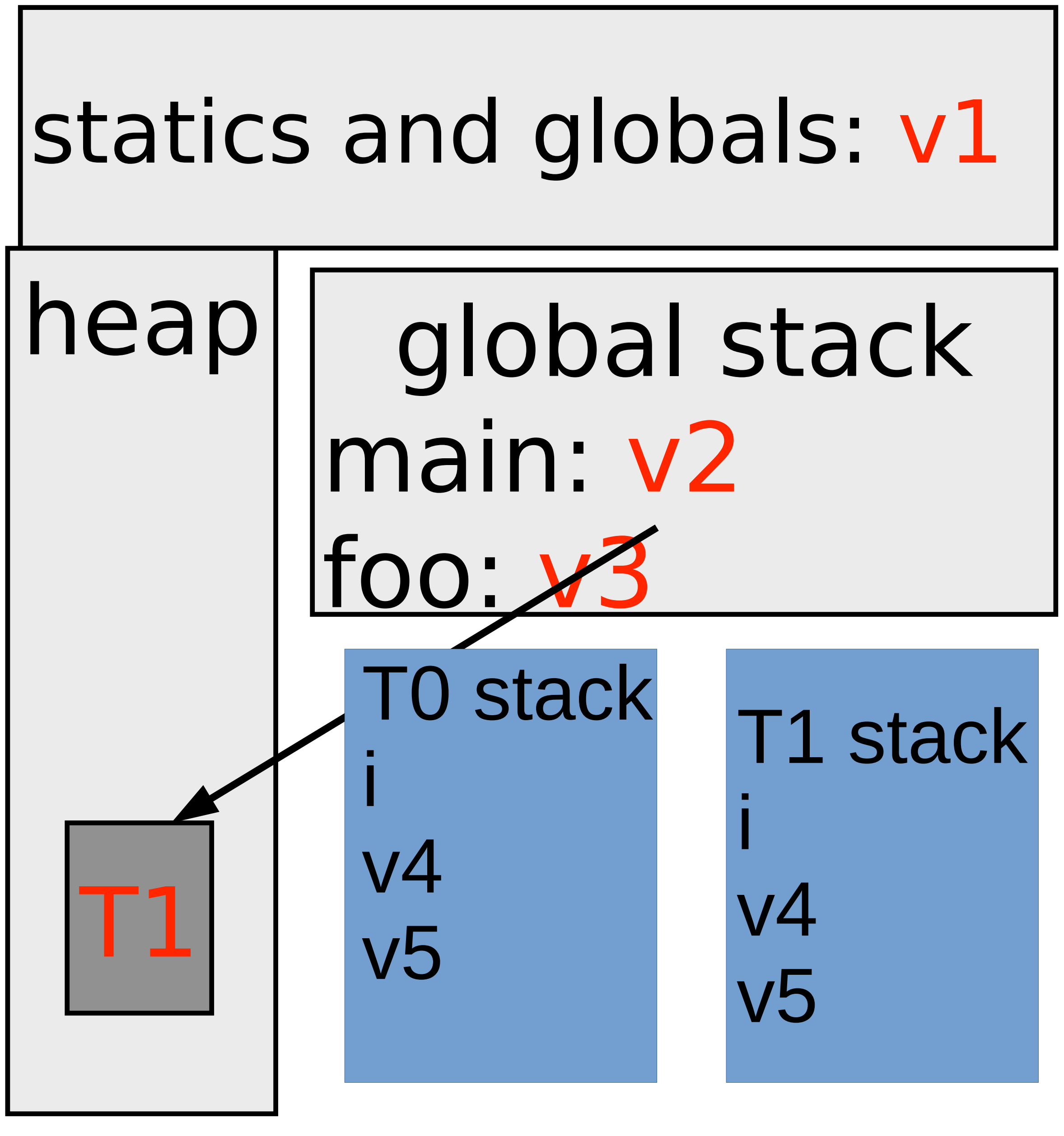
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for
    for (int i=0; i < n; i++) {
        int v4;
        T1 *v5 = malloc(sizeof(T1));
}}
```



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));
}}
```

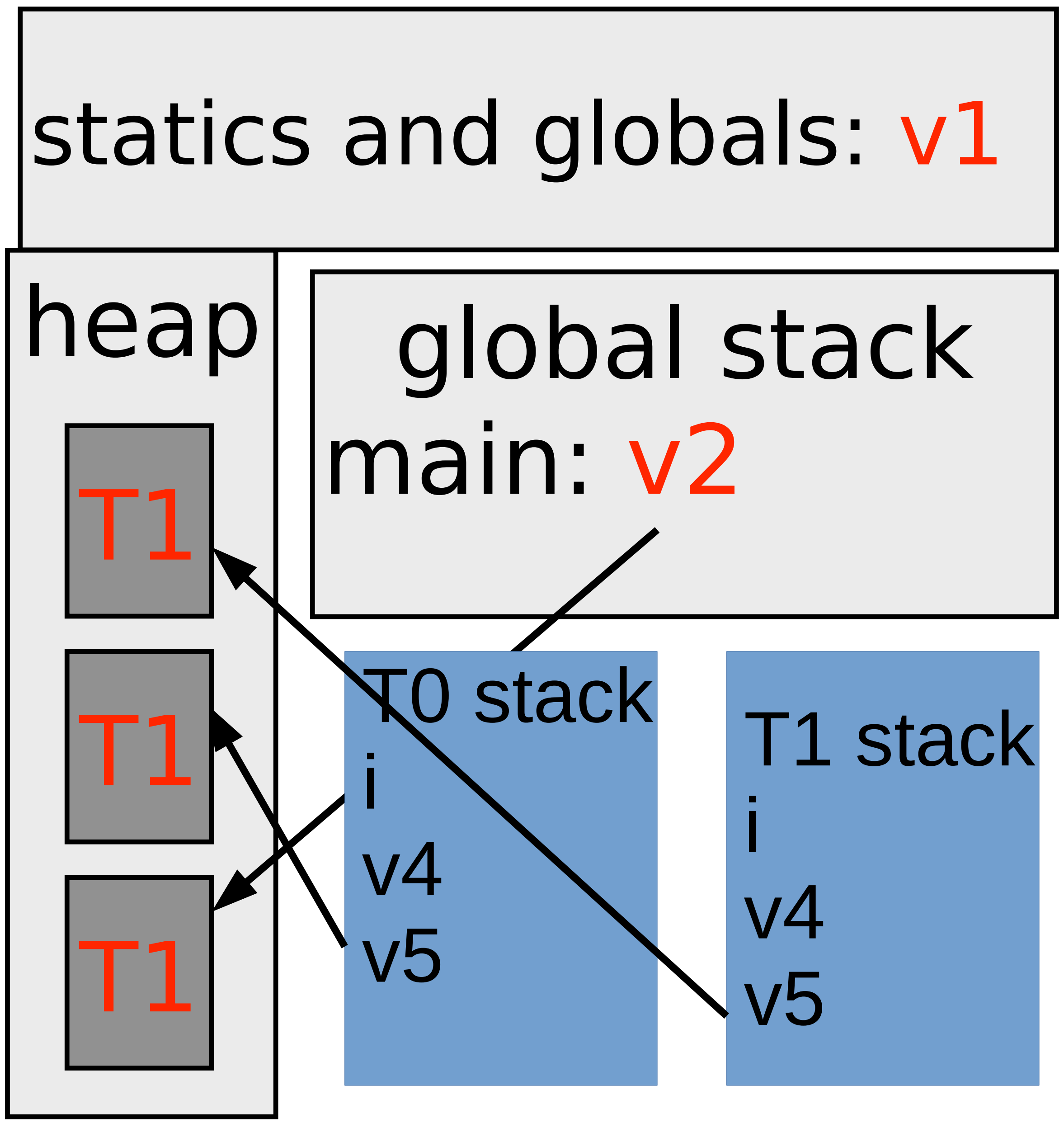


Note private loop index variables. OpenMP automatically makes the **parallel** *loop index private*

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

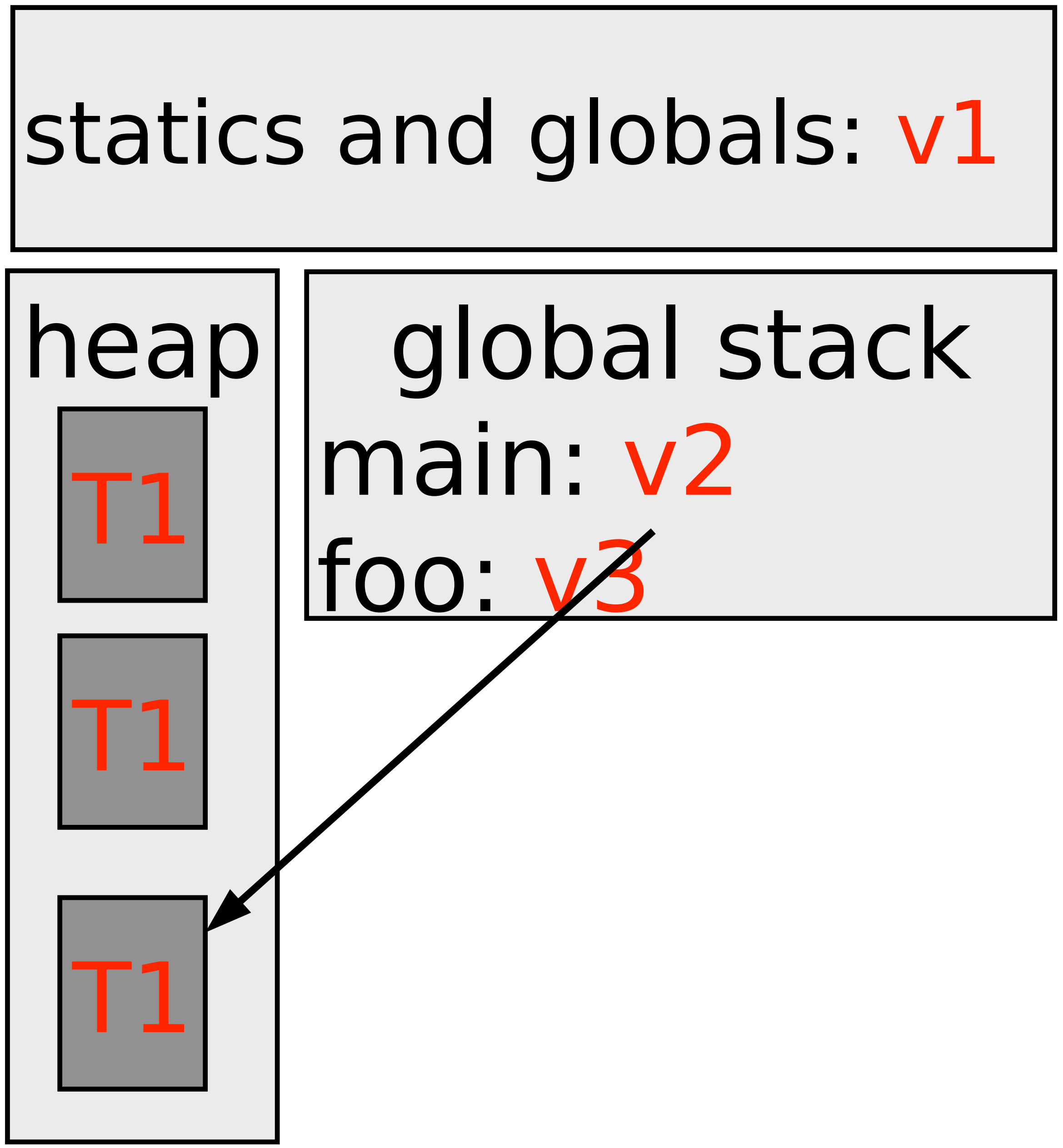
```
int v1;
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for
   for (i=0; i < n; i++) {
      int v4;
      T1 *v5 = malloc(sizeof(T1));
}}
```



Context after *parallel for* finishes

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

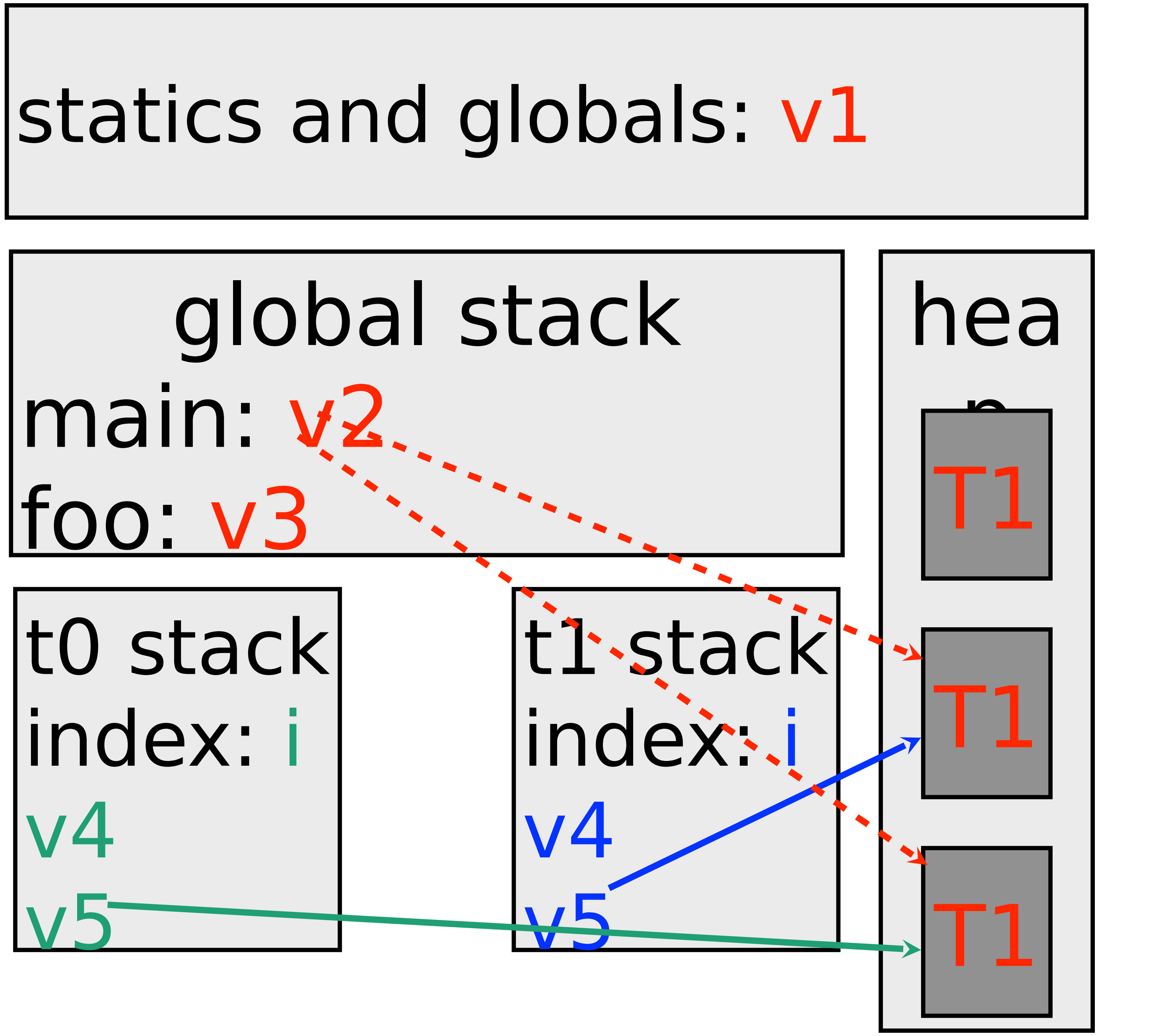
```
int v1;
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for

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



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

v2 points to one of the T2 objects that was allocated.
Which one? It depends.

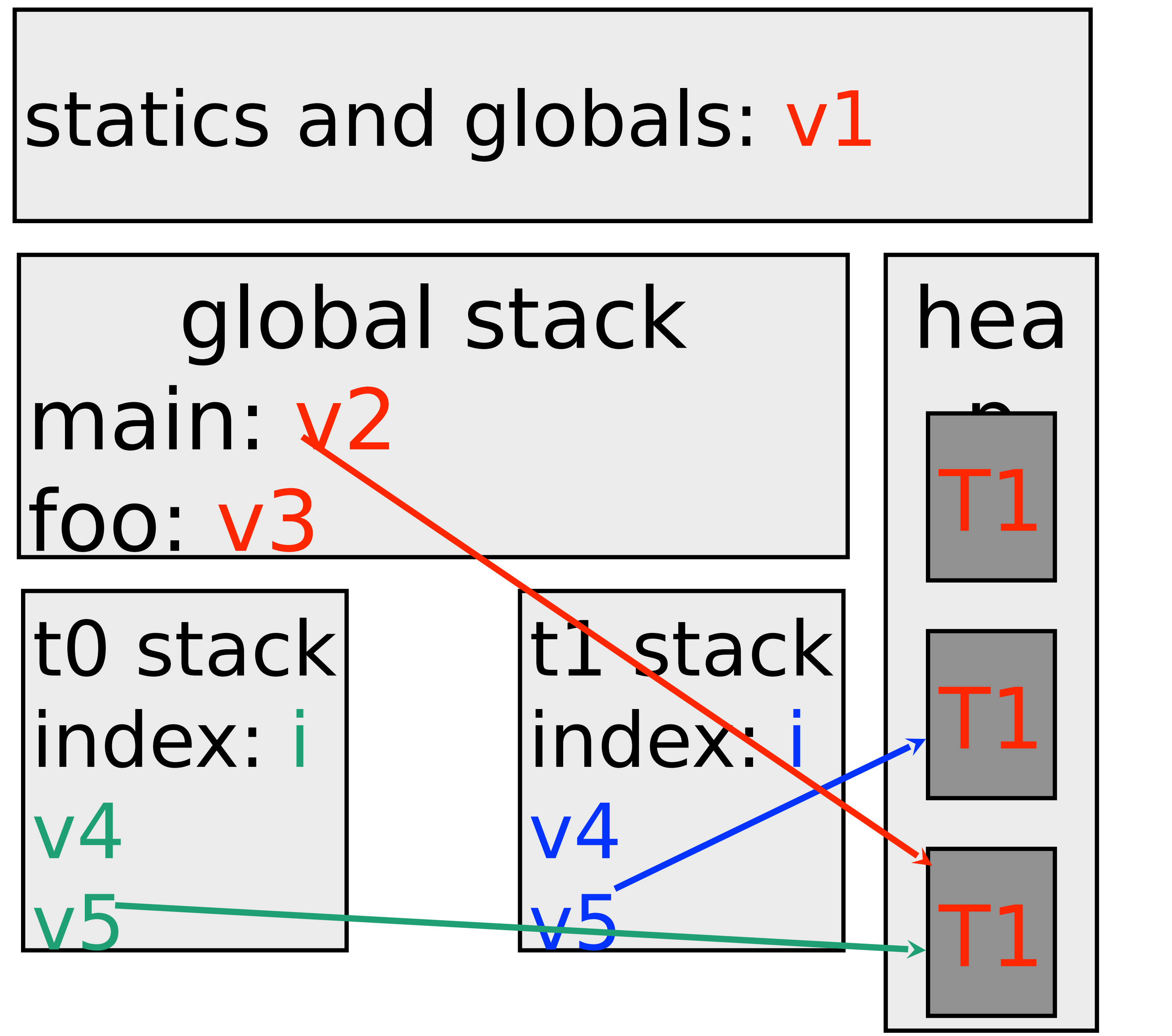
```
int v1;
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for
   for (i=0; i < n; i++) {
      int v4;
      T1 *v5 = malloc(sizeof(T1));
      v2 = (T1) v5
}}
```



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

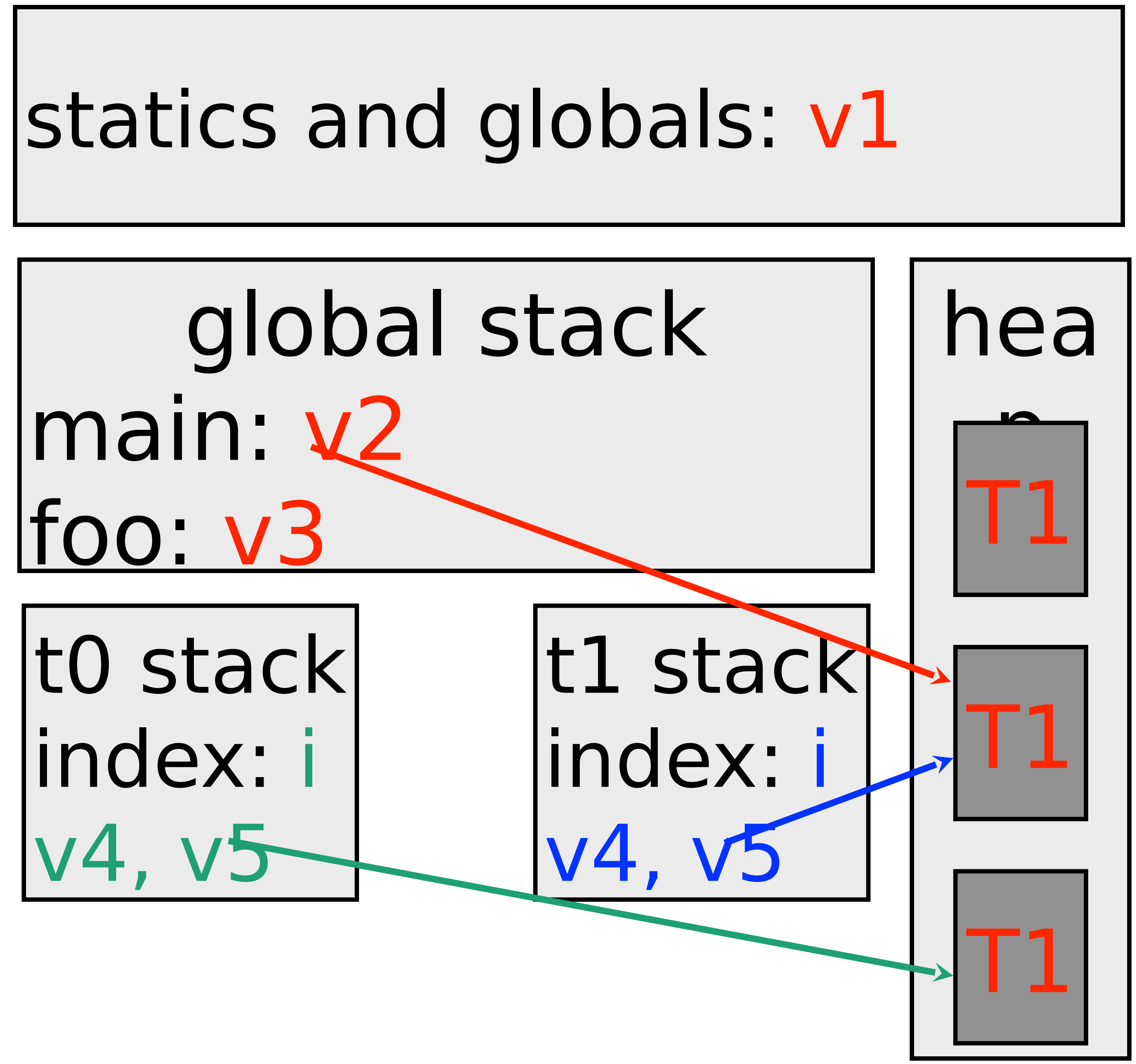
```
int v1;
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for
   for (i=0; i < n; i++) {
      int v4;
      T1 *v5 = malloc(sizeof(T1));
      v2 = (T1) v5
}}
```



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;
      T1 *v5 = malloc(sizeof(T1));
      v2 = (T1) v5
}}
```



Three (possible) problems with this code

First – do we care which object **v2** points to?

```
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(T2));
      v2 = (T1) v5
}}
```

Second – there is a *race* on **v2**

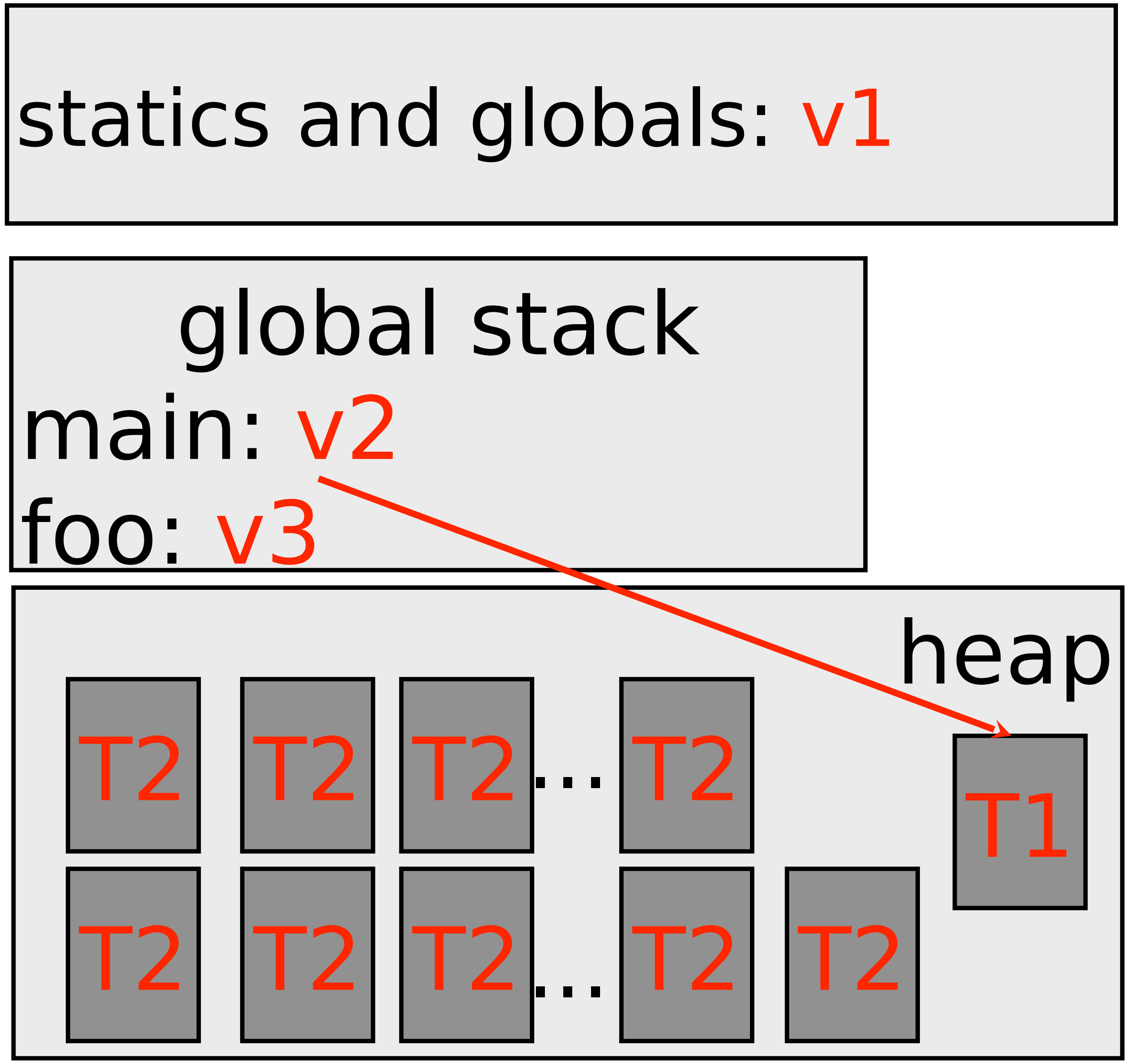
Two threads write to **v2**, but there is no intervening synchronization

Races are very bad – don't do them!

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;
        T2 *v5 = malloc(sizeof(T2));
}}
```



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]);
   }
}
```

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

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.

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.

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

```
#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]);
   }
}
```

What if we don't want a private variable?

- 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 of tmp 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 $i = n-1$

```
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 $i=n-1$. What happens if a thread other than the one executing iteration $i=n-1$ 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
- We want 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 the loop, 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 – Let’s use a *t* shared across threads

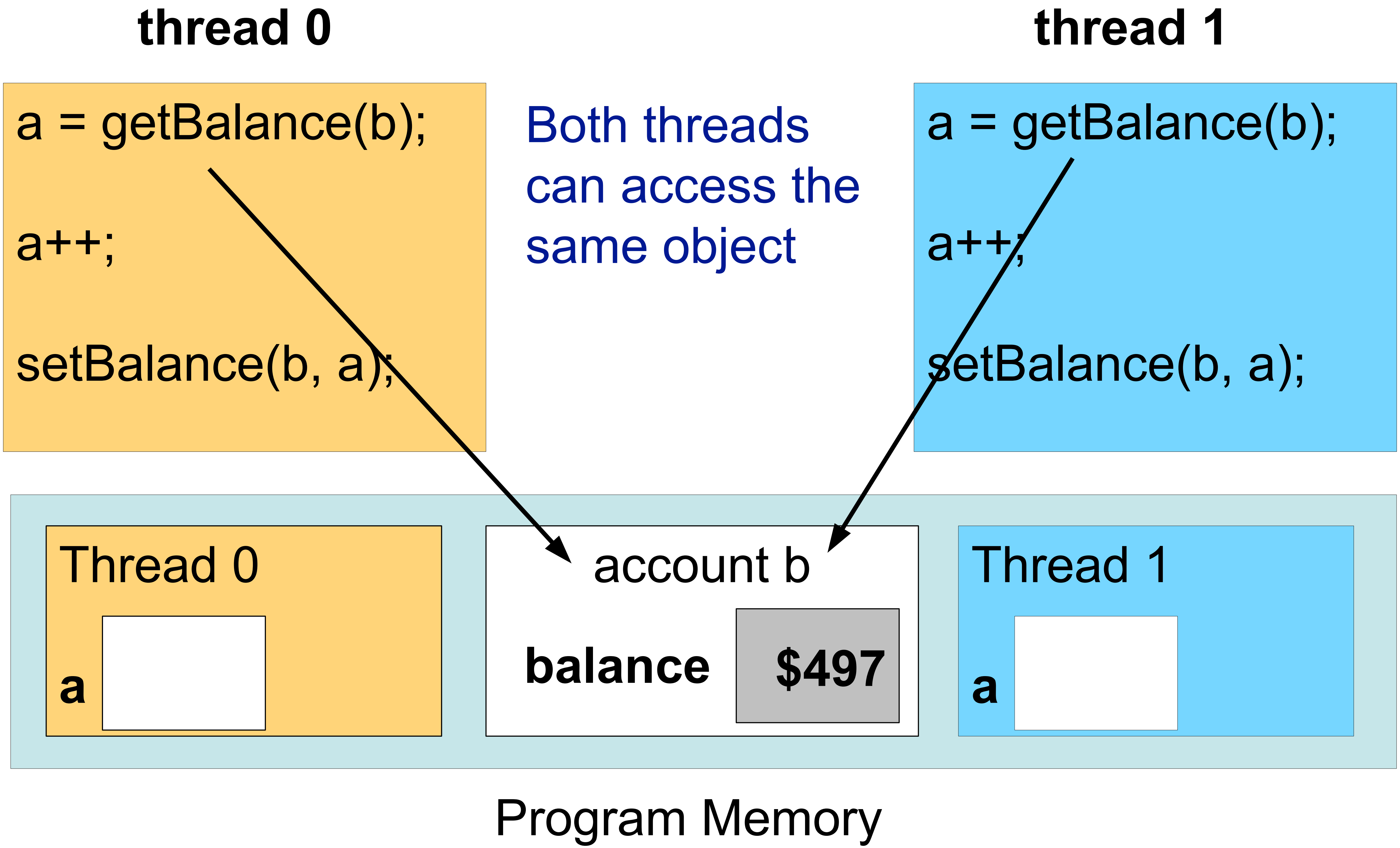
```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
   t += a[i];
}
t = t/n
```

What is wrong with this?

Need to execute `t+= a[i];` *atomically*!

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

ordering and *atomicity* are important and different

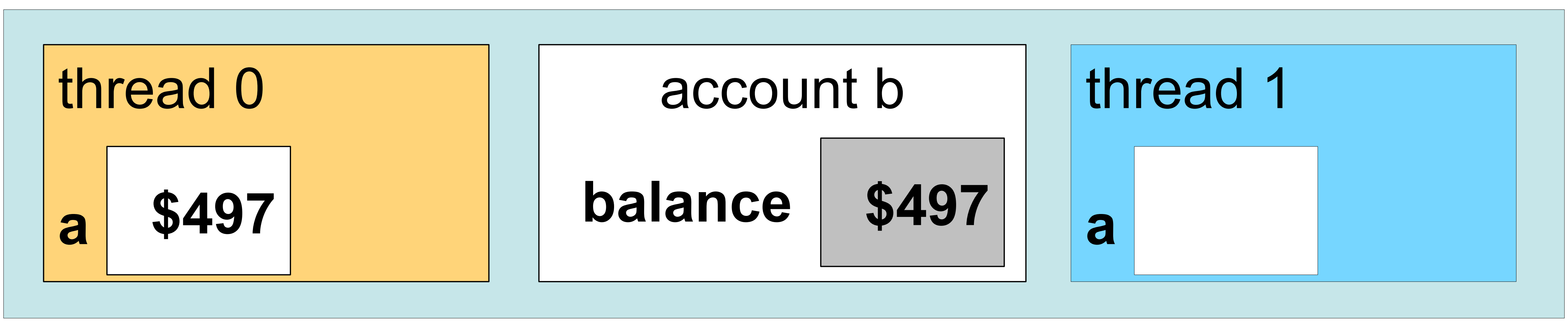


thread 0

```
a = getBalance(b);

a++;

setBalance(b, a);
```

thread 1

```
a = getBalance(b);

a++;

setBalance(b, a);
```



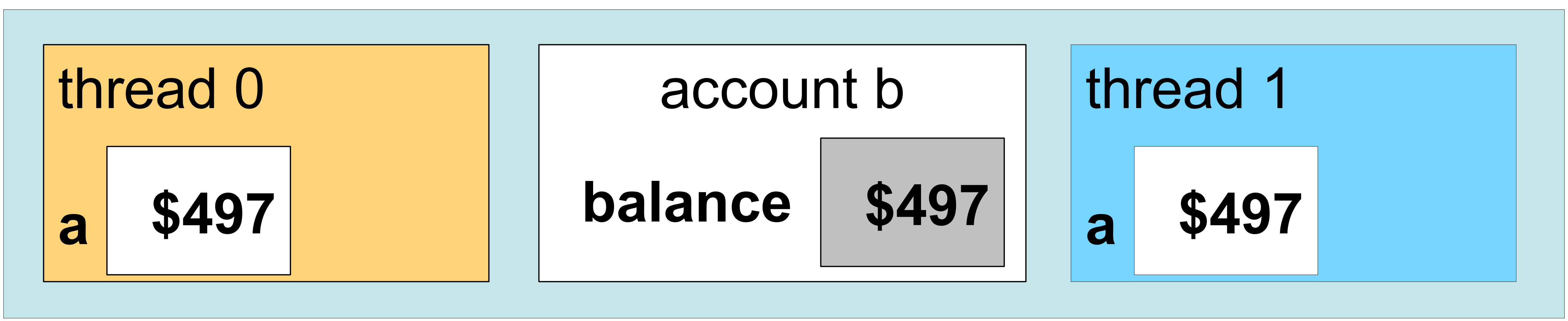
Program Memory

thread 0

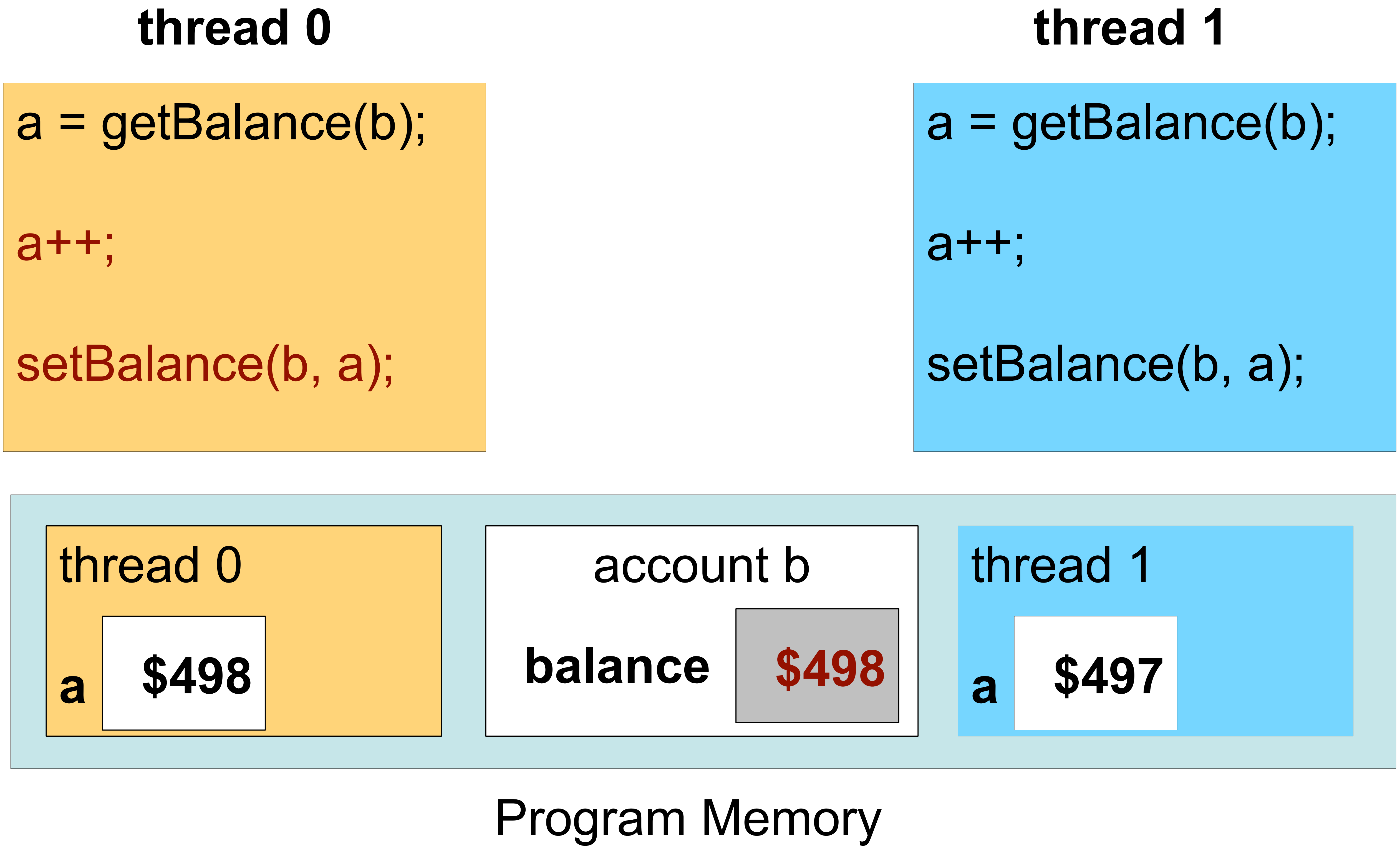
```
a = getBalance(b);

a++;

setBalance(b, a);
```

thread 1

```
a = getBalance(b);

a++;

setBalance(b, a);
```



Program Memory


thread 0
thread 1
a = getBalance(b);
a++;
setBalance(b, a);
a = getBalance(b);
a++;
setBalance(b, a);
thread 0
a
$498
account b
balance
$498
thread 1
a
$497
Program Memory

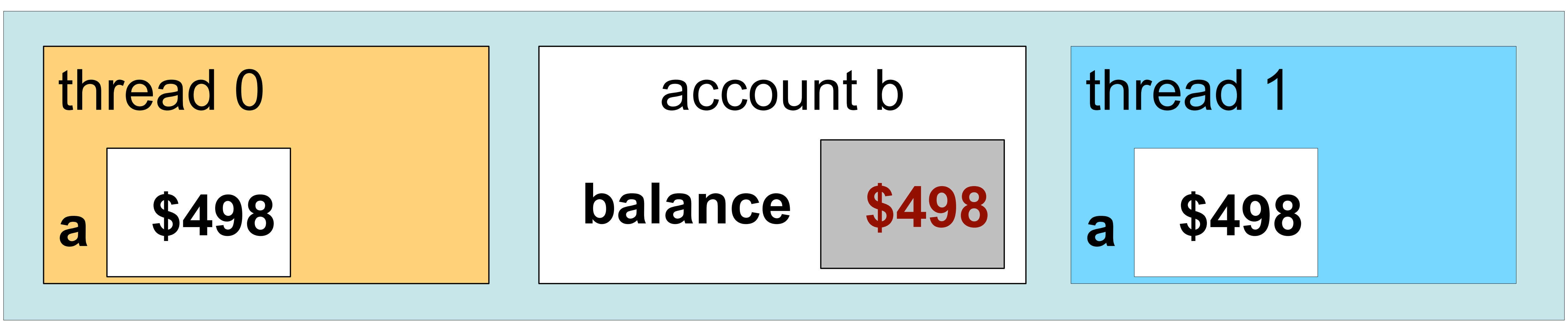
thread 0

```
a = getBalance(b);

a++;

setBalance(b, a);
```

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

thread 1

```
a = getBalance(b);

a++;

setBalance(b, a);
```



Program Memory

synchronization enforces atomicity

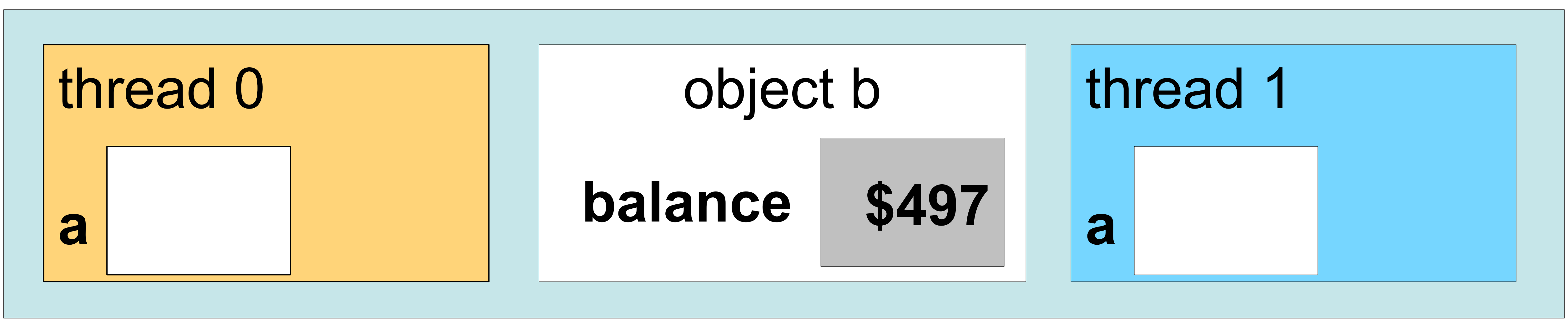
thread 0

```
#pragma omp critical {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

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

thread 1

```
#pragma omp critical {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```



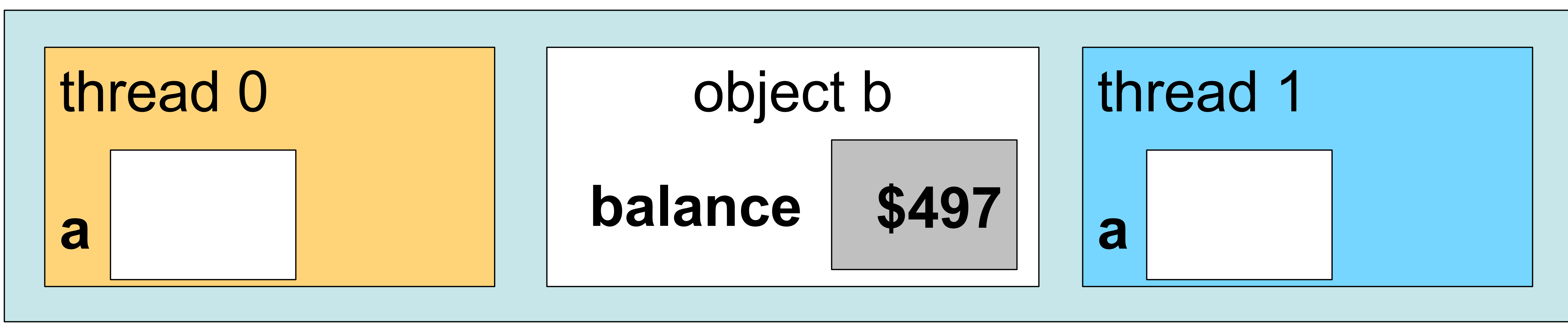
Program Memory

One thread acquires the *lock*

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the *lock* is free

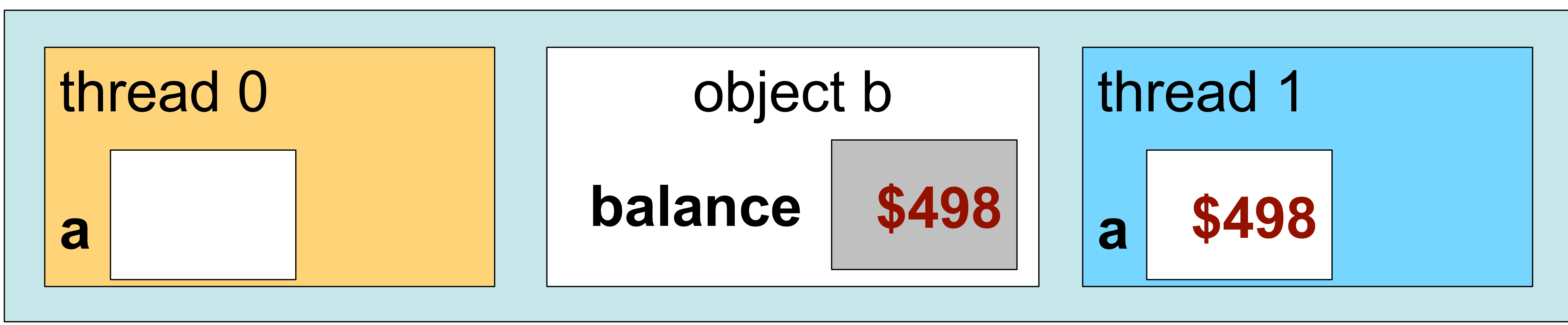


One thread acquires the lock

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free

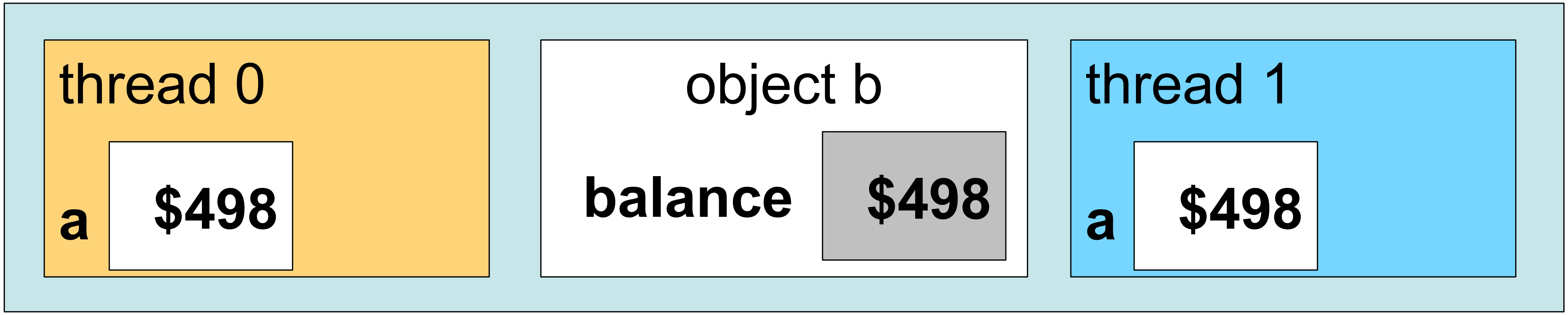


One thread acquires the lock

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free

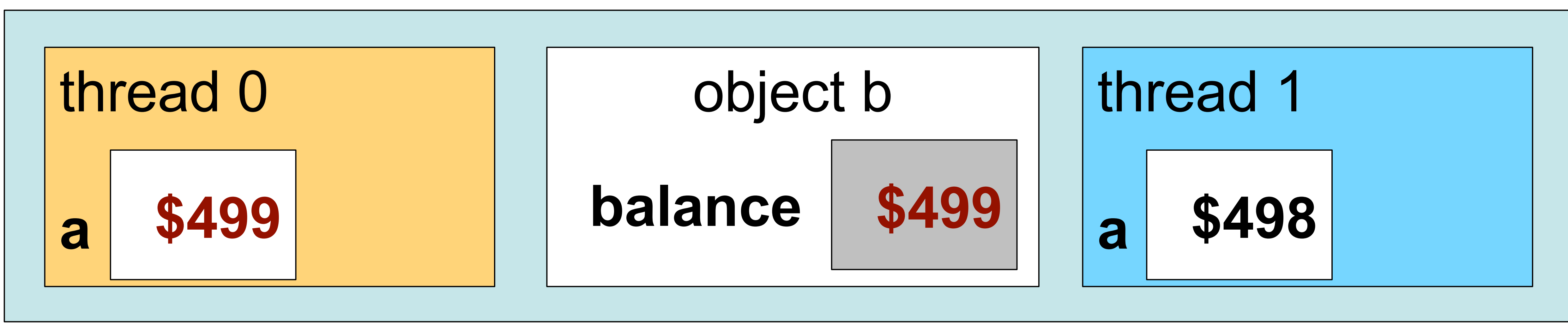


One thread acquires the lock

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free



Locks typically do not enforce ordering

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

Either order is possible

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

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
#omp critical
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

- Same thing as in the bank example can happen with our program
  - 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.
- The first update of t is lost.

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

Third (and correct, but too slow) attempt

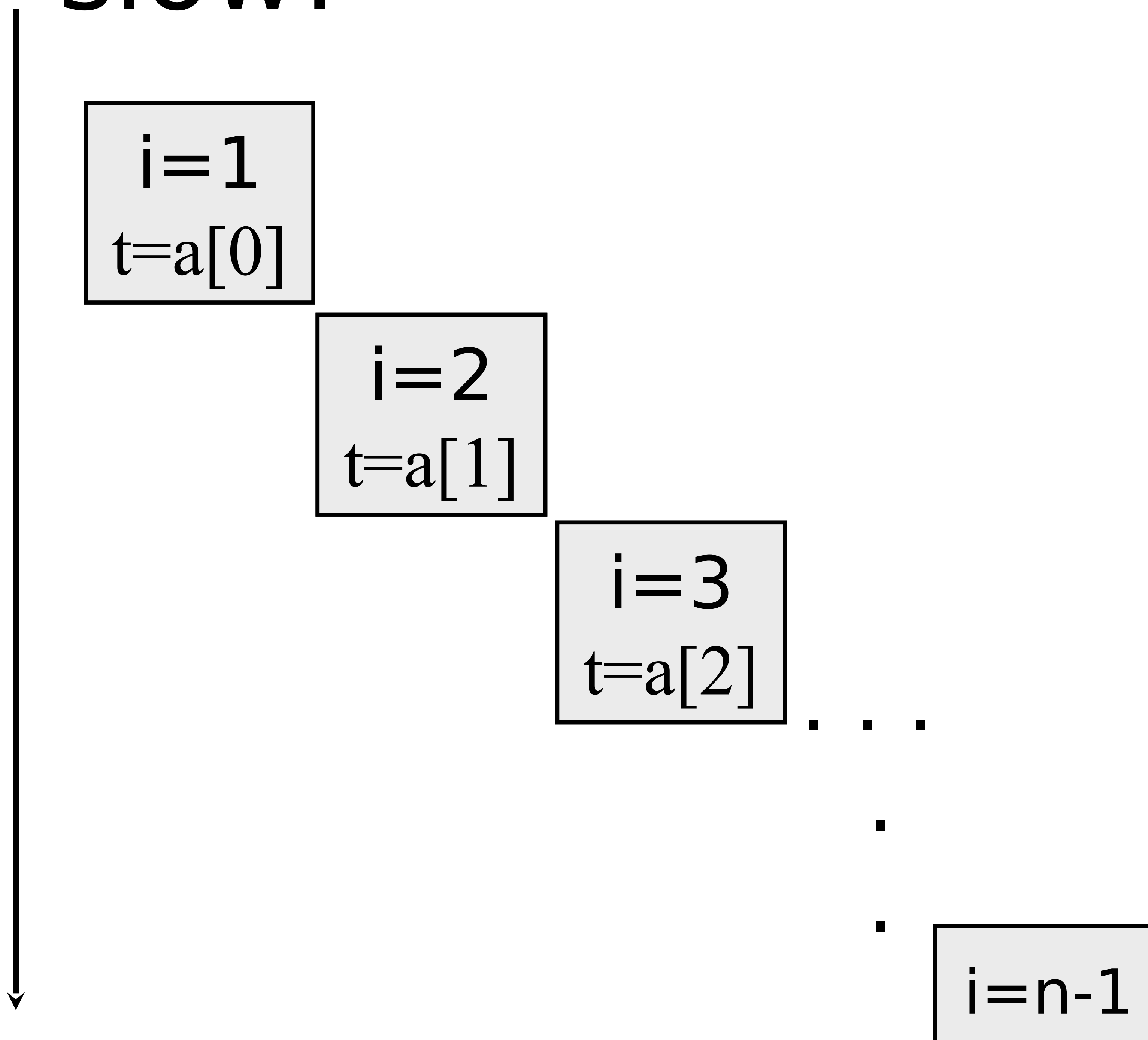
- use a *critical* section in the code
- executes the following (possible compound) statement atomically

```
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?

It is effectively serial, and too slow!

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

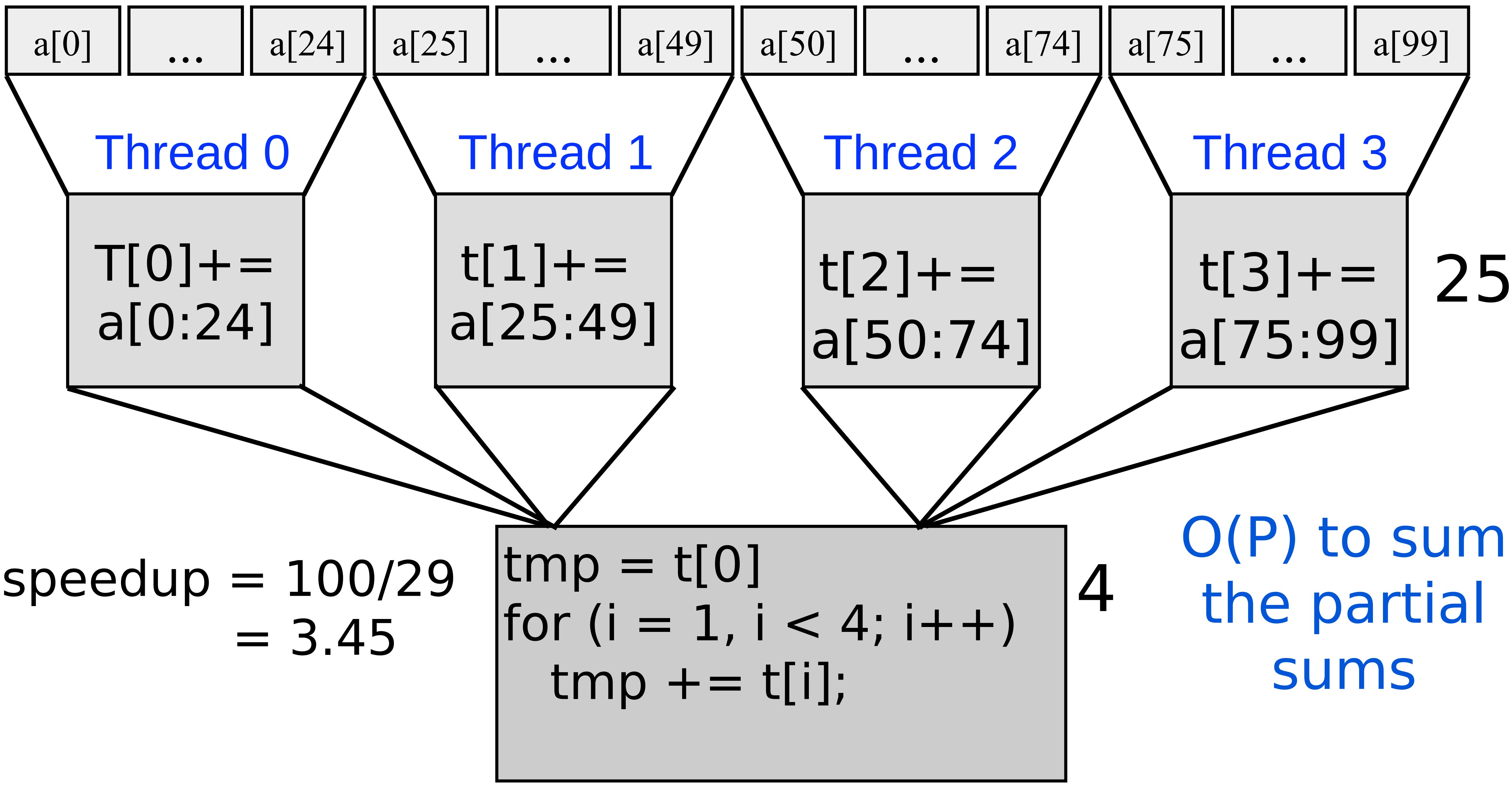


time = O(n)

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



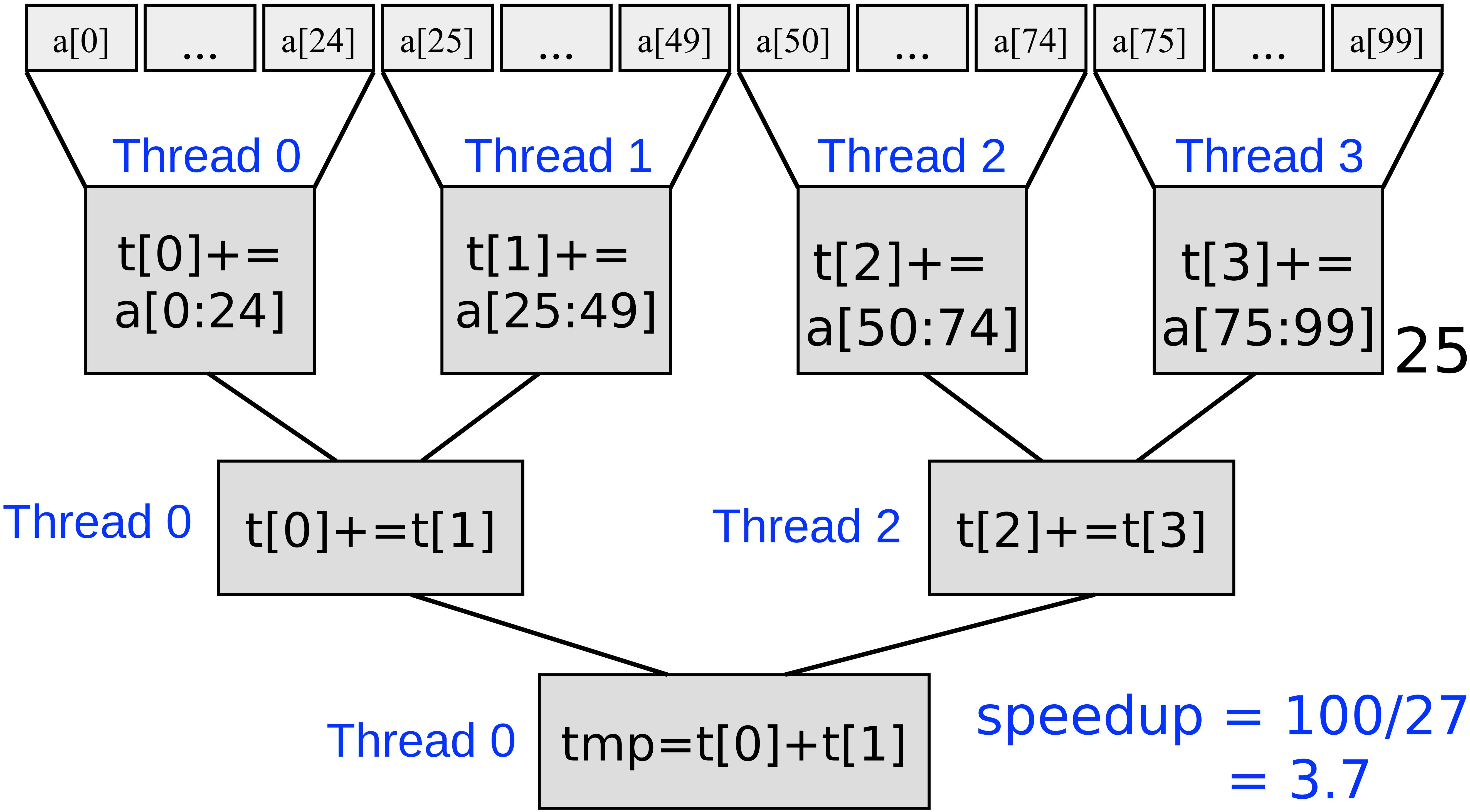
How can we do this in OpenMP?

```
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;
```

parallel
serial
OpenMP function

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

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?

Sequential:

i = 1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
t_1 = 1	t_1 = 3	t_1 = 6	t_1 = 10	t_1 = 15	t_1 = 21	t_1 = 28	t_1 = 36

Parallel:

i = 1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
t_1 = 1	t_1 = 3	t_1 = 6	t_1 = 10	t_1 = 5	t_1 = 11	t_1 = 18	t_1 = 26
Thread = 0				Thread = 1			

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

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*

Static

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

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

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

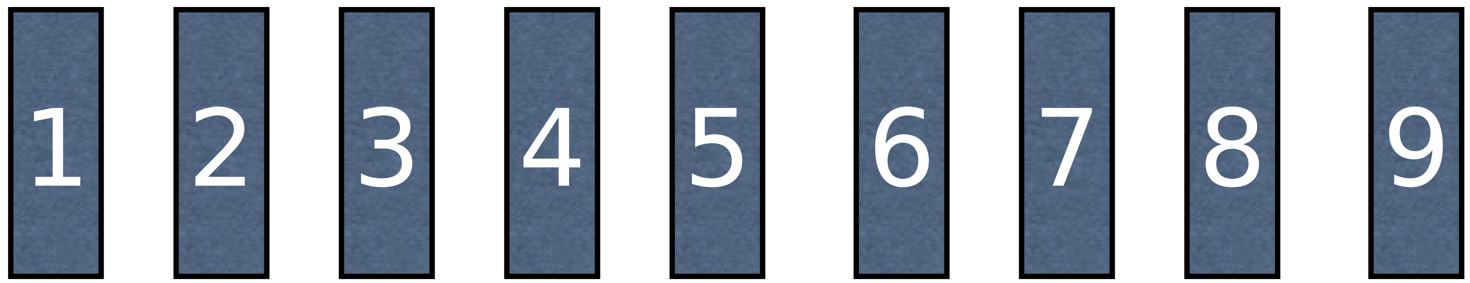
The schedule clause - dynamic

- 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
  - As threads need work, they are given additional *chunk* iterations of work

The schedule clause – guided

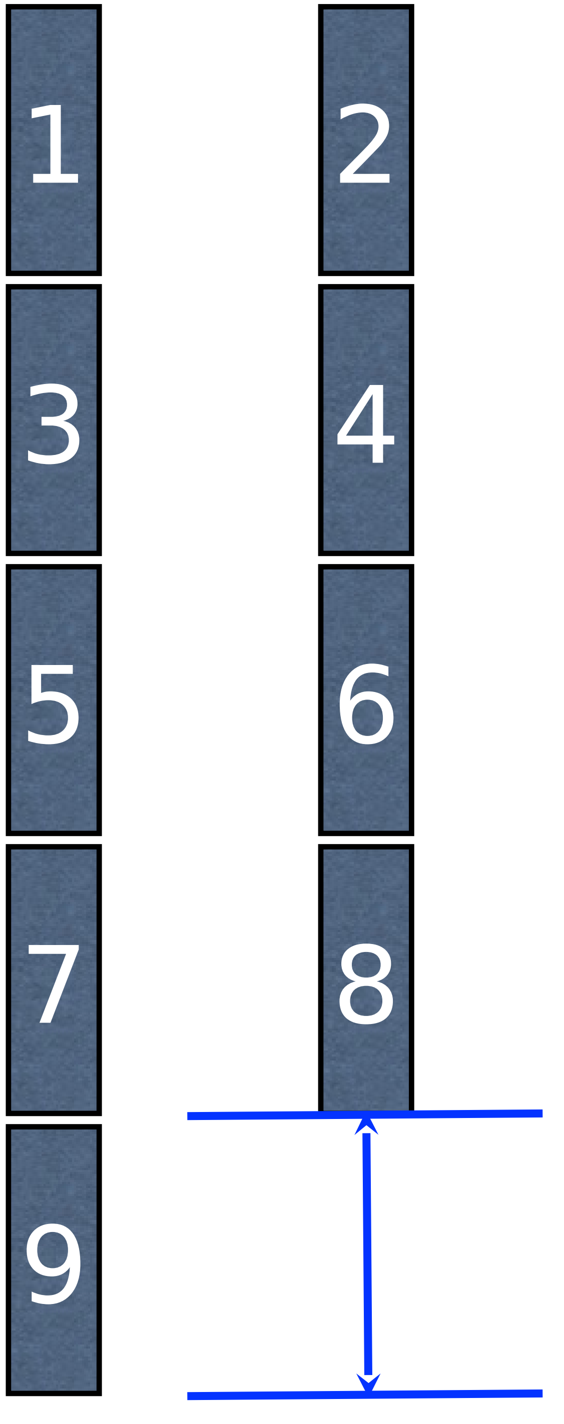
- schedule(*type[, chunk])*) (type [,chunk]) is
  - (guided,chunk): uses *guided self scheduling* heuristic. Starts with big chunks and decreases to a minimum chunk size of *chunk*
  - runtime - type depends on value of OMP_SCHEDULE environment variable, e.g. setenv OMP_SCHEDULE="static,1"

Guided with two threads example



Dynamic schedule with large blocks

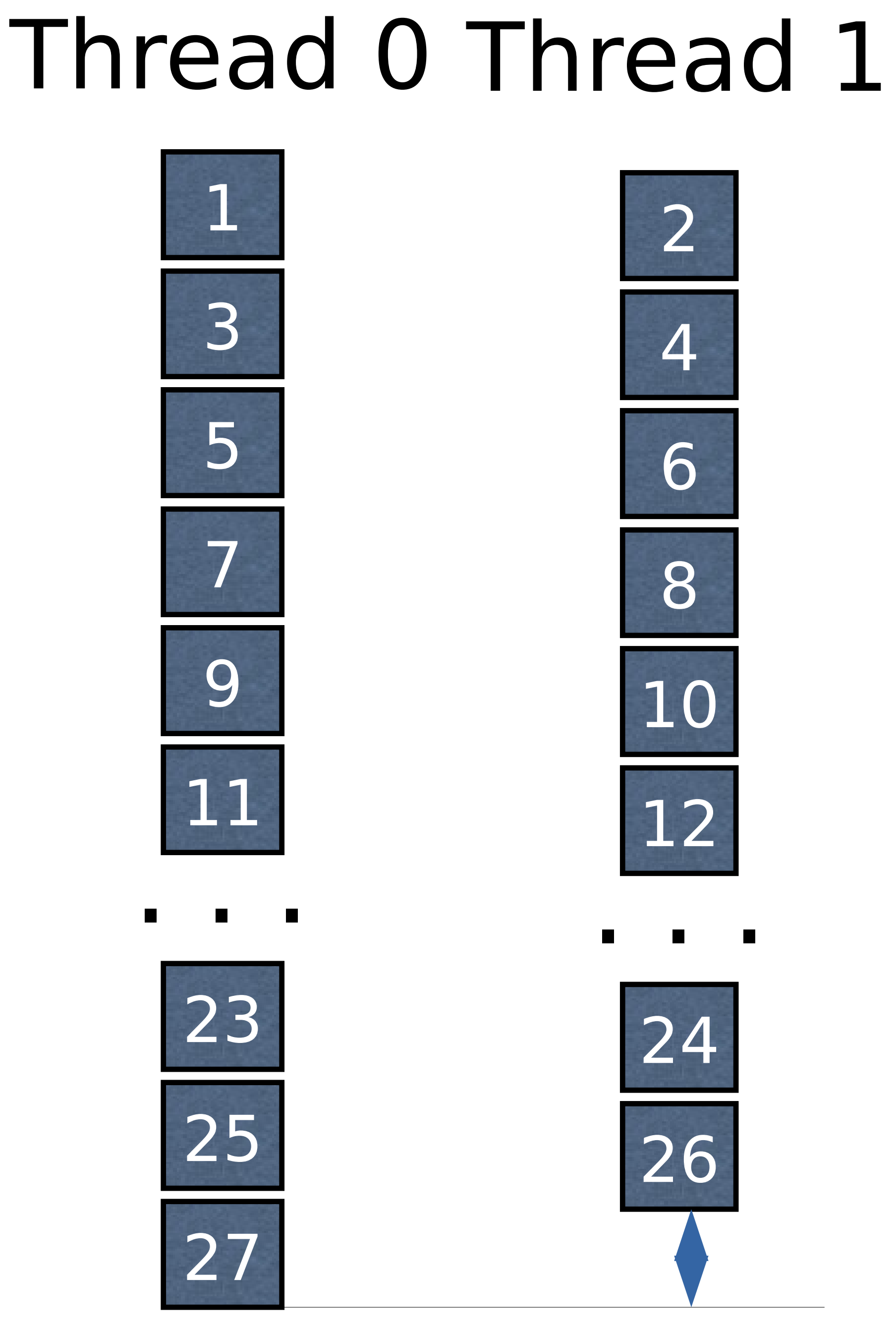
Large blocks reduce scheduling costs, but lead to large load imbalance



Dynamic schedule with small blocks

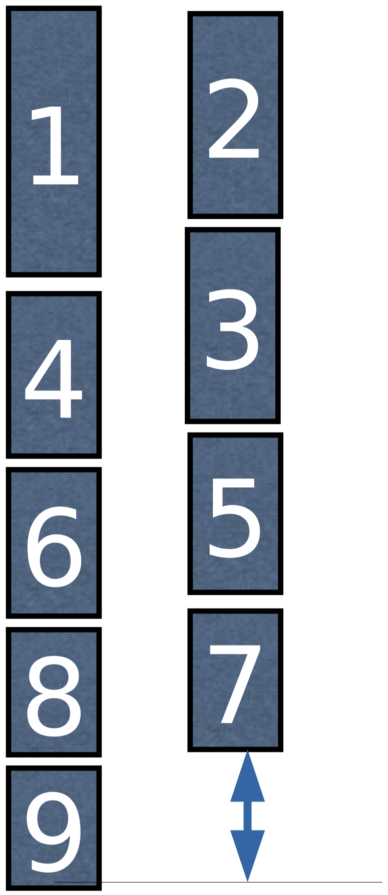
Small blocks have a smaller load imbalance, but with higher scheduling costs.

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.

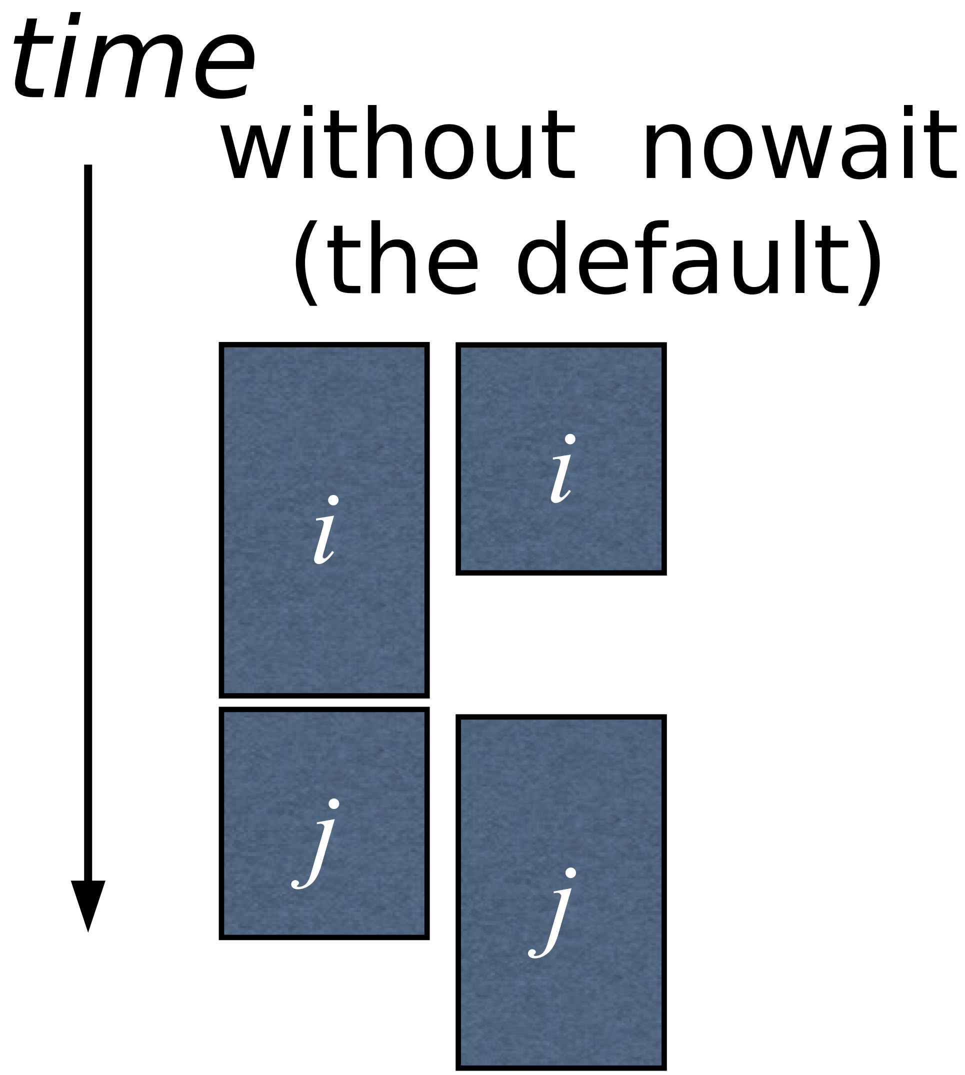


The nowait clause

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

barrier here

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



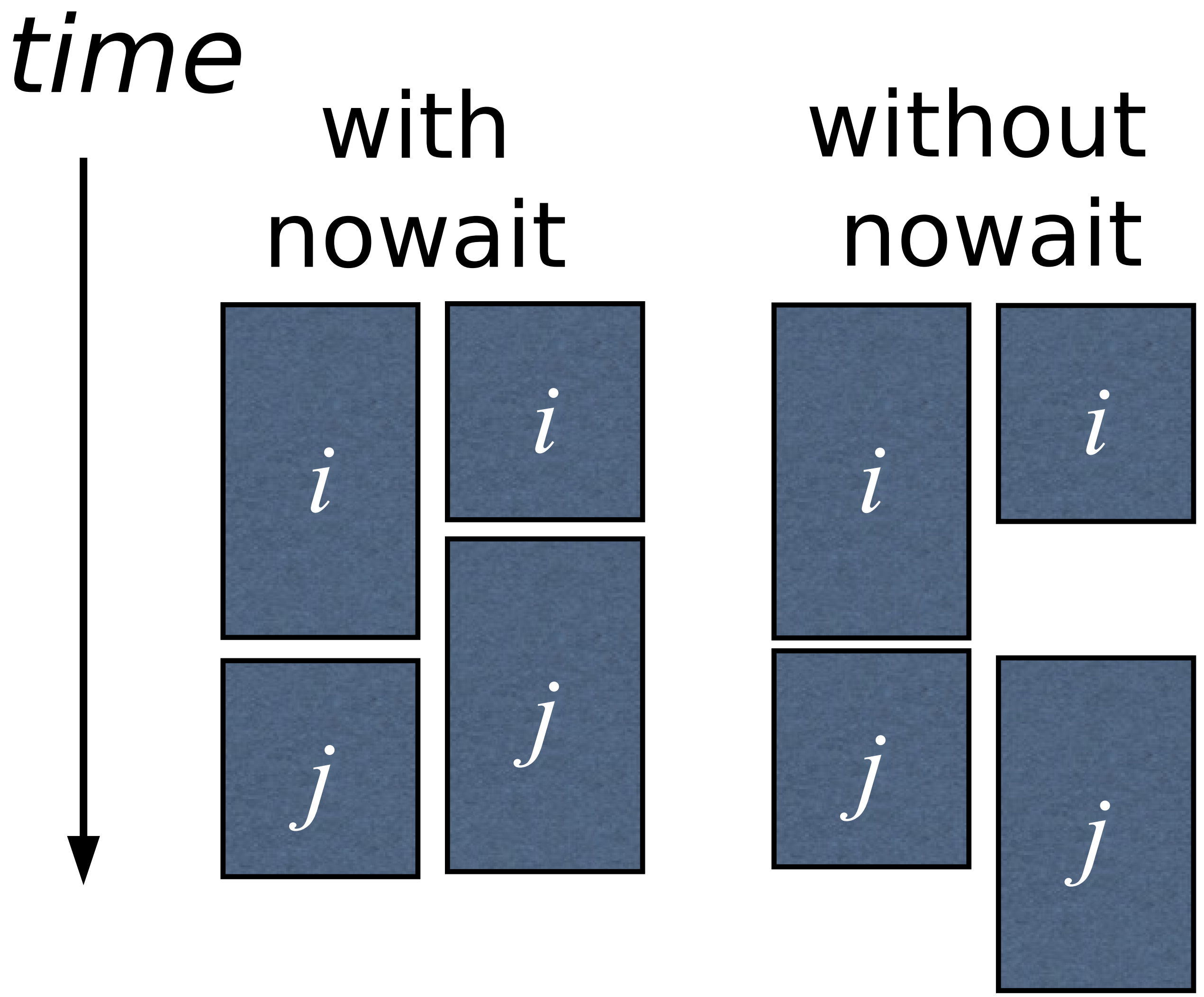
Only the static distribution with the same bounds guarantees the same thread will execute the same iterations from both loops.

The nowait clause

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

NO barrier here

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

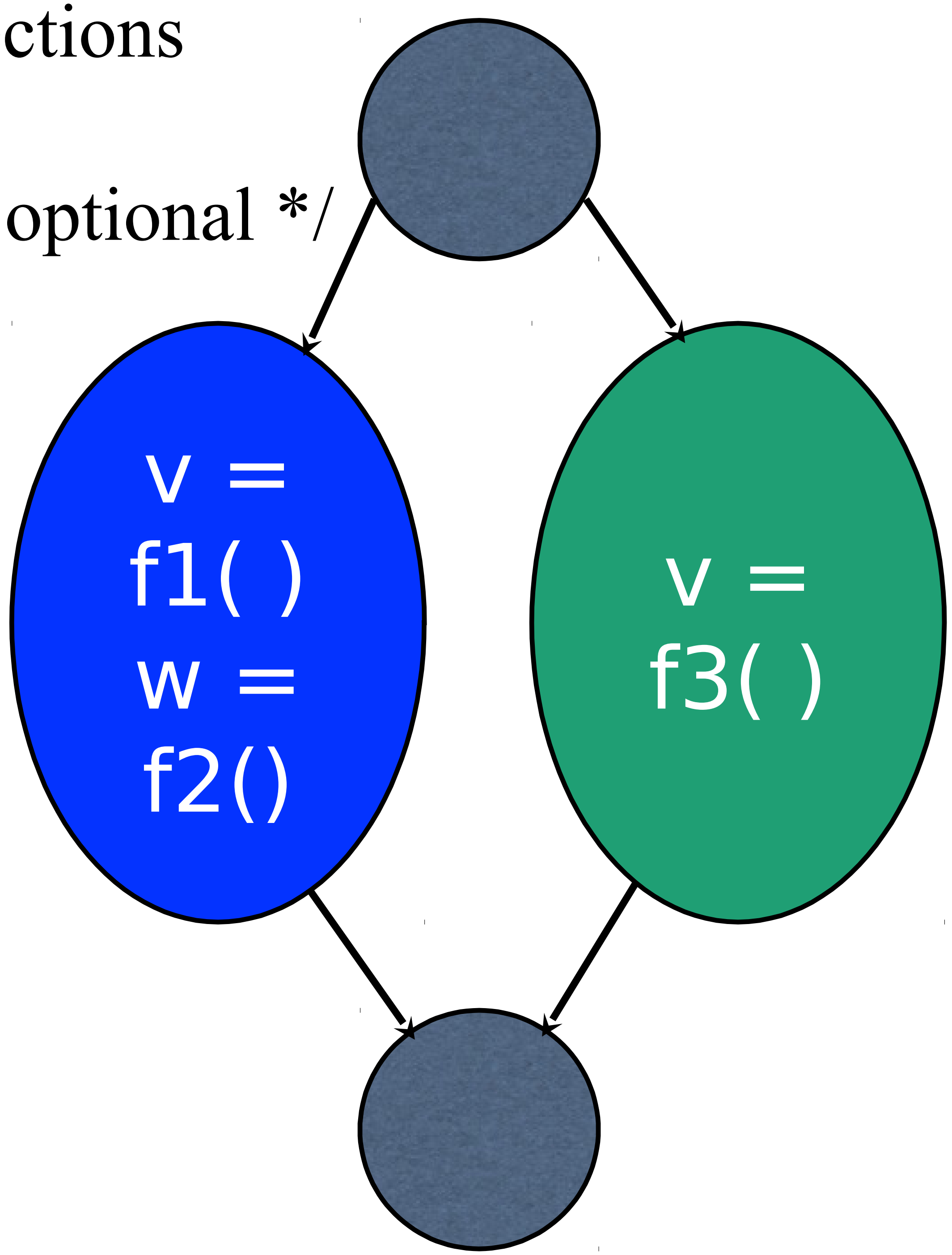


Only the static distribution with the same bounds guarantees the same thread will execute the same iterations from both loops.

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

- There is parallelism across useful work in the example because independent and different work pulled off of the queue 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 can be useful in a parallel region.

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");
}
```

Differs from critical in that critical lets the statement execute on every thread executing the parallel region, but one at a time.

Requires statement following the pragma to be executed by a single thread.

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

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]
}
```

- 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.*

Program Translation for Microtasking Scheme

```
   Subroutine x
   ...
C$OMP PARALLEL DO
   DO j=1,n
     a(j)=b(j)
   ENDDO
   …
   END
```

→

```
subroutine x
…
call scheduler(1,n,a,b,loopsub)
…
END
```

```
subroutine loopsub(lb,ub,a,b)
integer lb,ub
DO jj=lb,ub
   a(jj)=b(jj)
ENDDO
END
```

How are loops scheduled?

- A work queue is maintained with work for threads to get
- An entry for an chunk of the loop, represented by loopsub, is something like:

int lb

int ub

ptr to a and b

A ptr to subroutine loopsub

- As each thread completes a work item, it grabs a work item from the queue, invokes the subroutine pointed to passing the other members of the struct as arguments.

Parallel ExecutionScheme

- Most widely used: Microtasking scheme