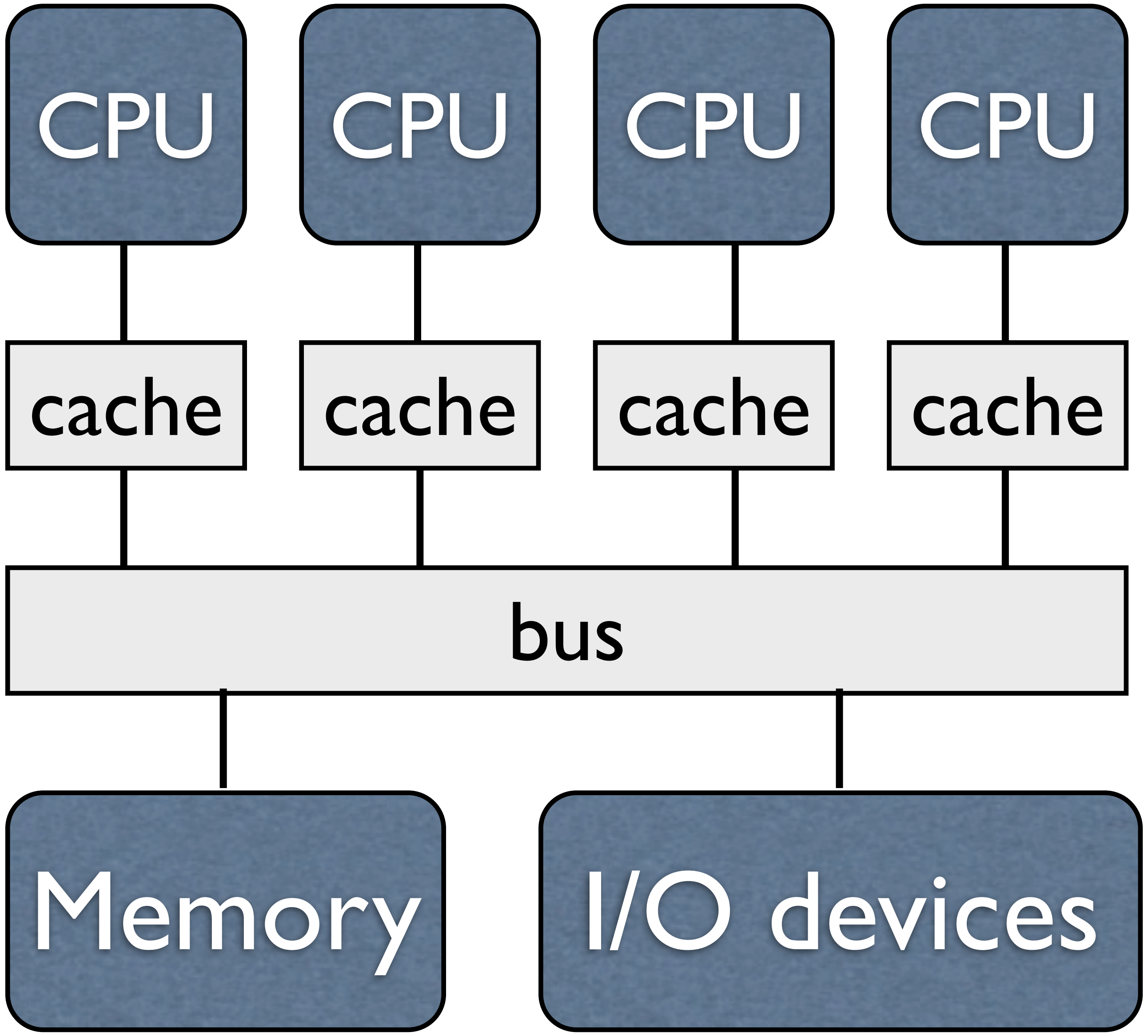
OpenMP

What is OpenMP

- An open standard for shared memory programming in C/C++ and Fortran
- supported by IBM, Intel, Gnu 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

OpenMP Hardware Model

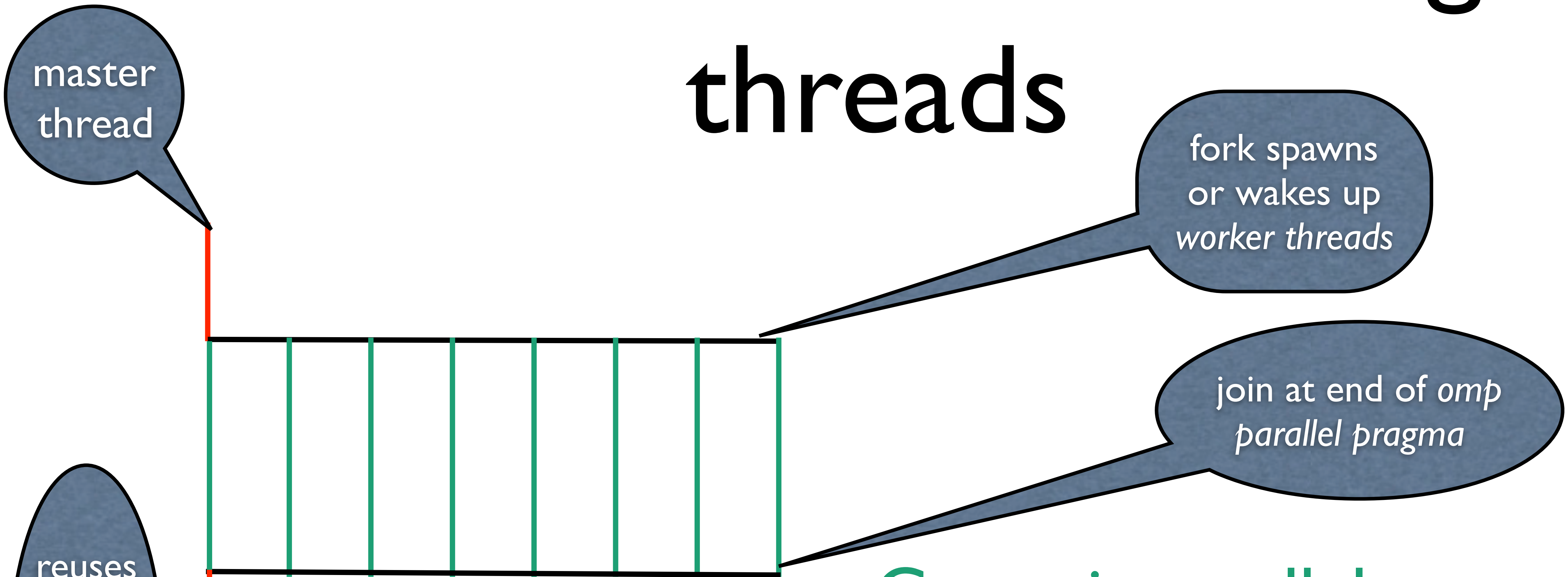
Uniform memory access shared memory machine is assumed

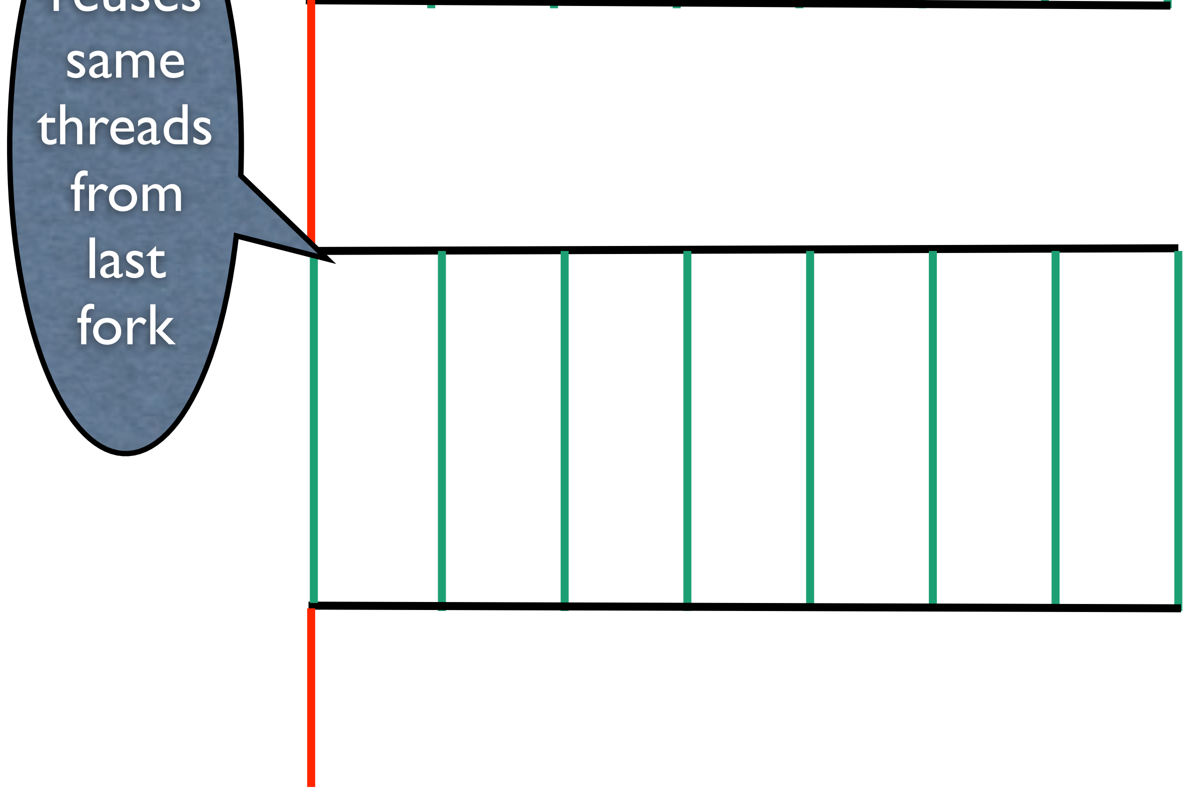


Fork/Join Parallelism

- 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 suppends the worker threads

Parallel execution using threads





Green is parallel execution

Red is sequential

For efficiency, worker threads are suspended, not killed, at the end of the execution

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

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:

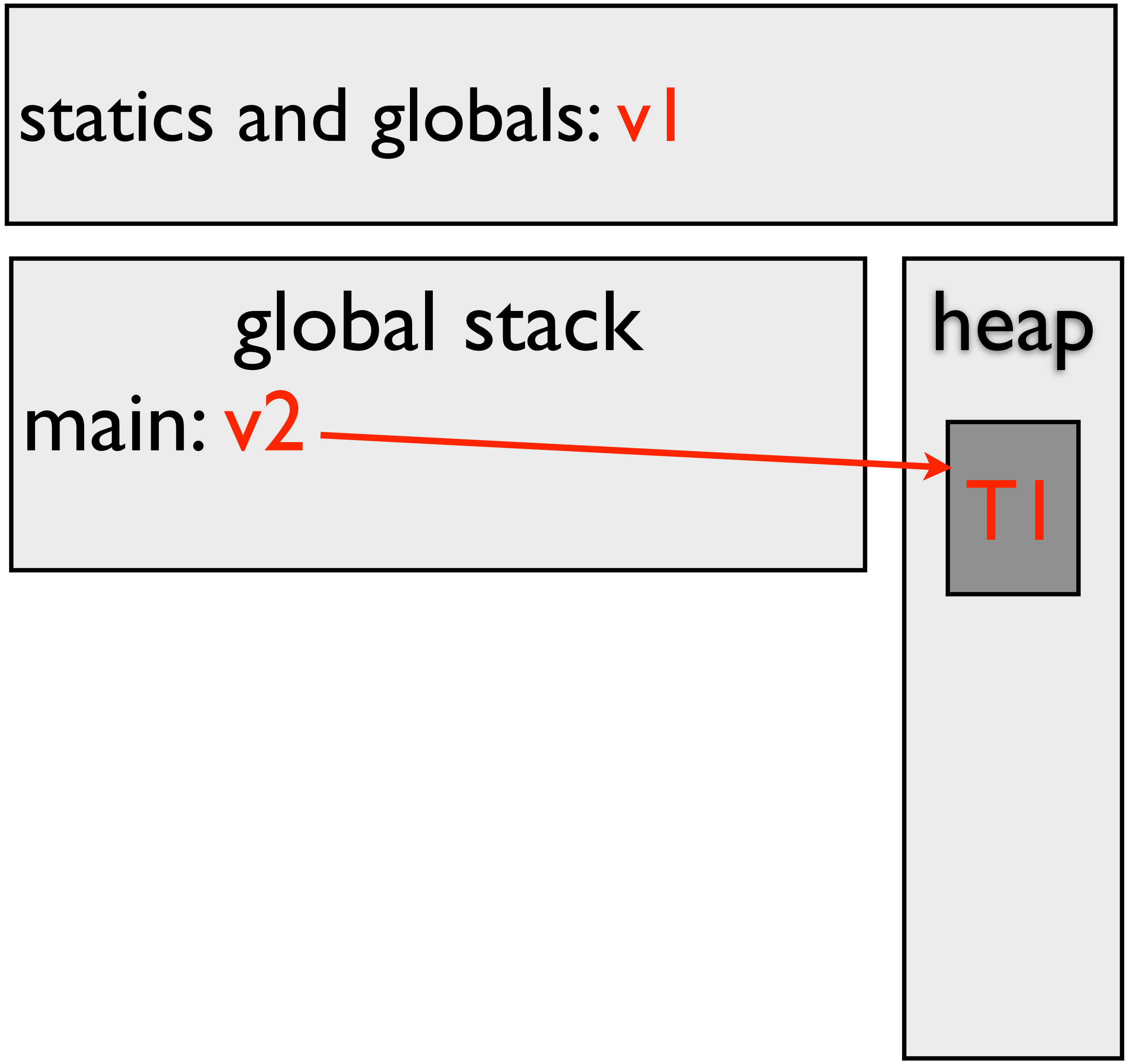
```
int v1;
...
main( ) {
   T1 *v2 = malloc(sizeof(T1));
   ...
   f1( );
}
void f1( ) {
   int v3;
#pragma omp parallel for private(v4,v5)
   for (i=0; i < n; i++) {
      int v4;
      T2 *v5 = malloc(sizeof(T2));
}}
```

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

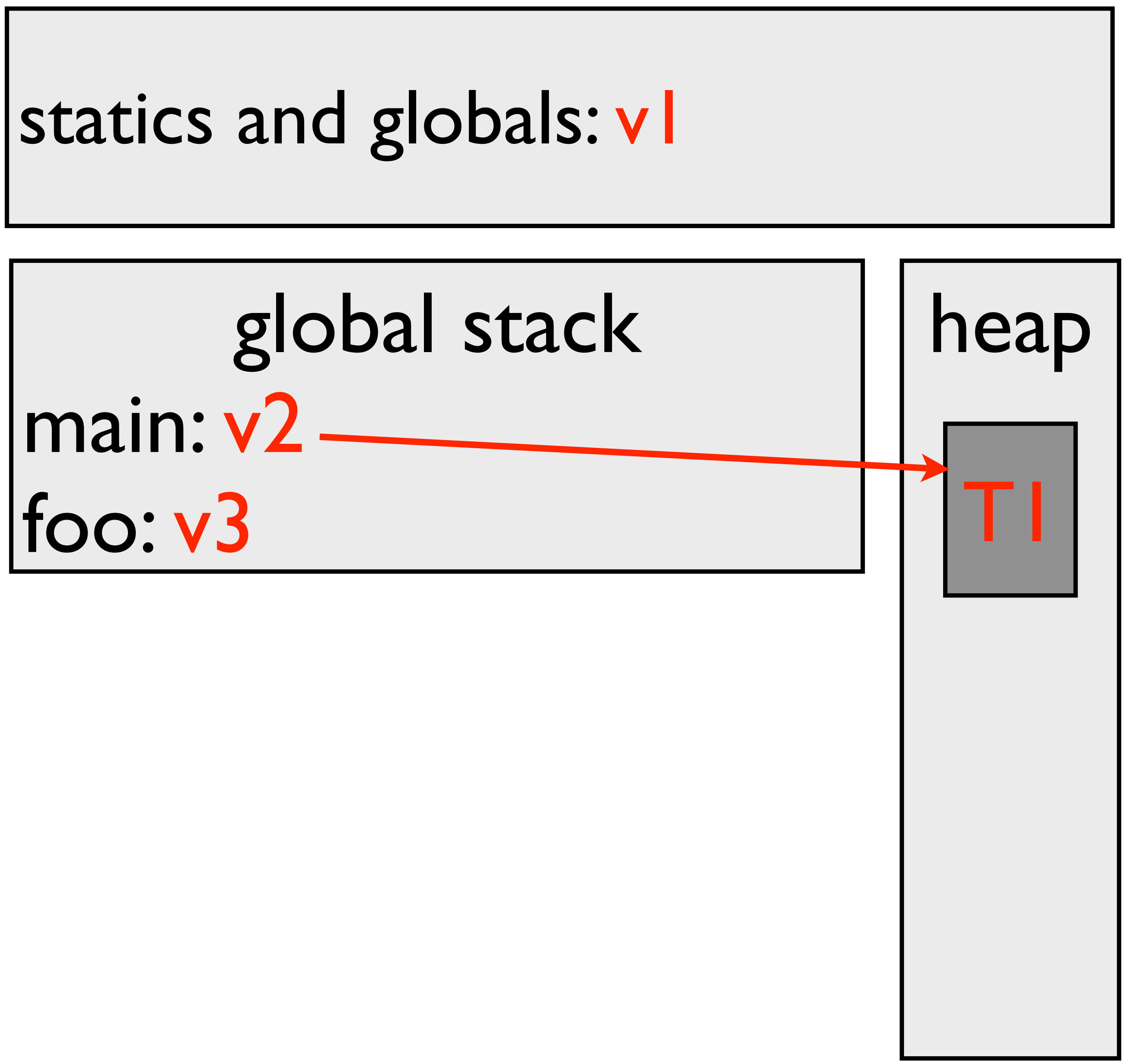
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for private(v4,v5)
    for (i=0; i < n; i++) {
        int v4;
        T2 *v5 = malloc(sizeof(T2));
}}
```



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

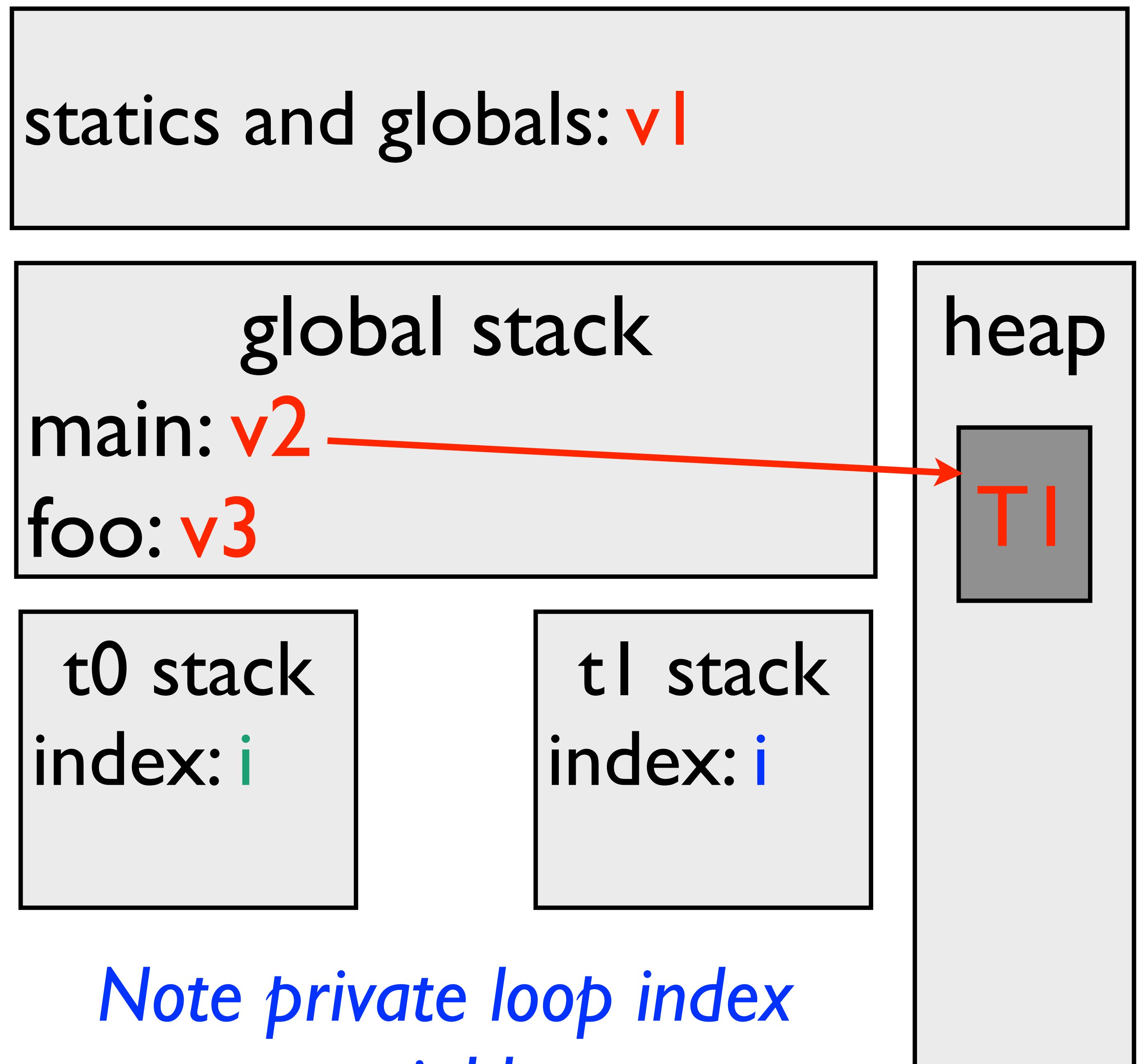
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for private(v4,v5)
    for (i=0; i < n; i++) {
        int v4;
        T2 *v5 = malloc(sizeof(T2));
}}
```



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



Note private loop index variables

Context after first iteration of the *parallel for*

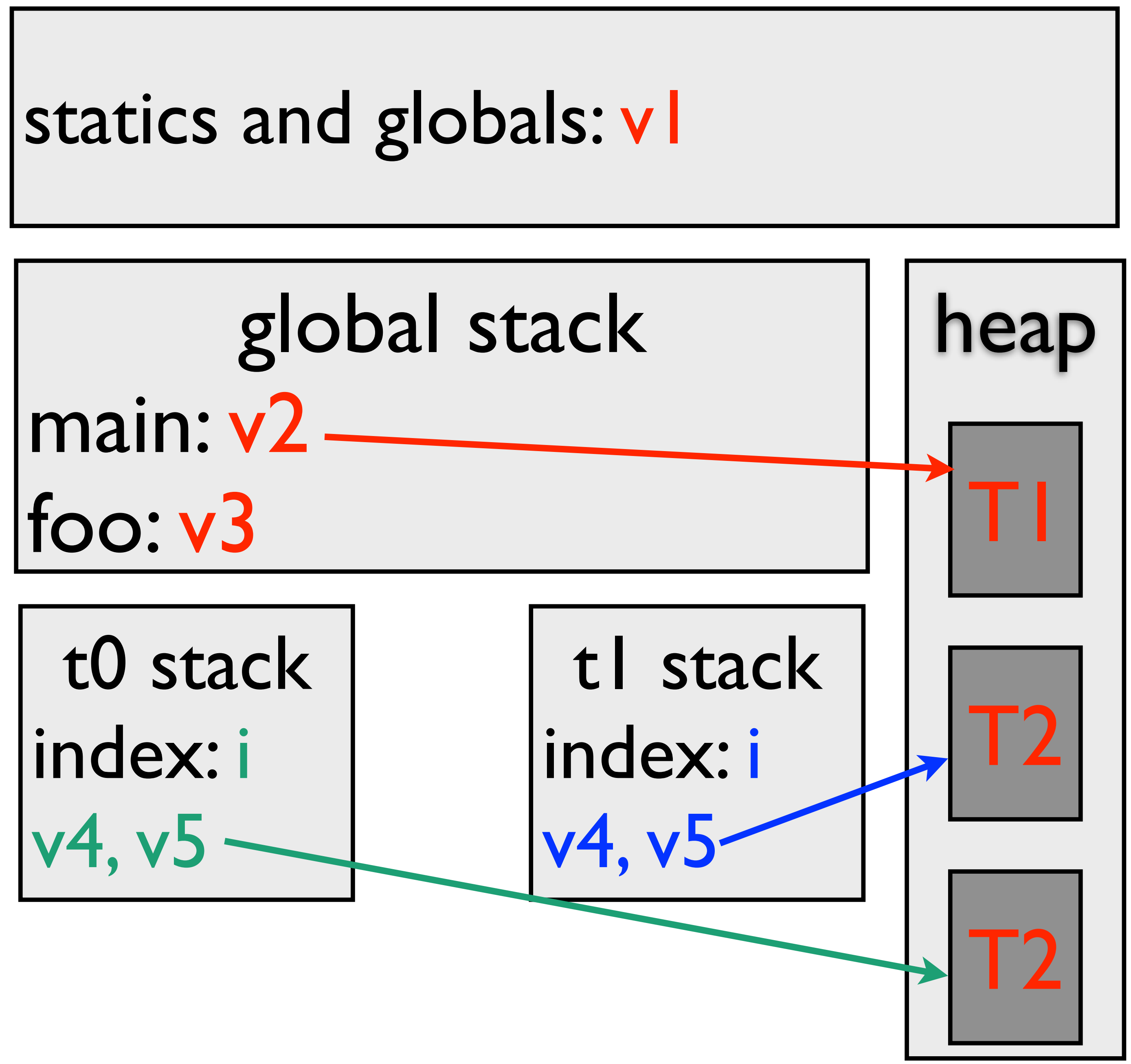
Storage, assuming two threads

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

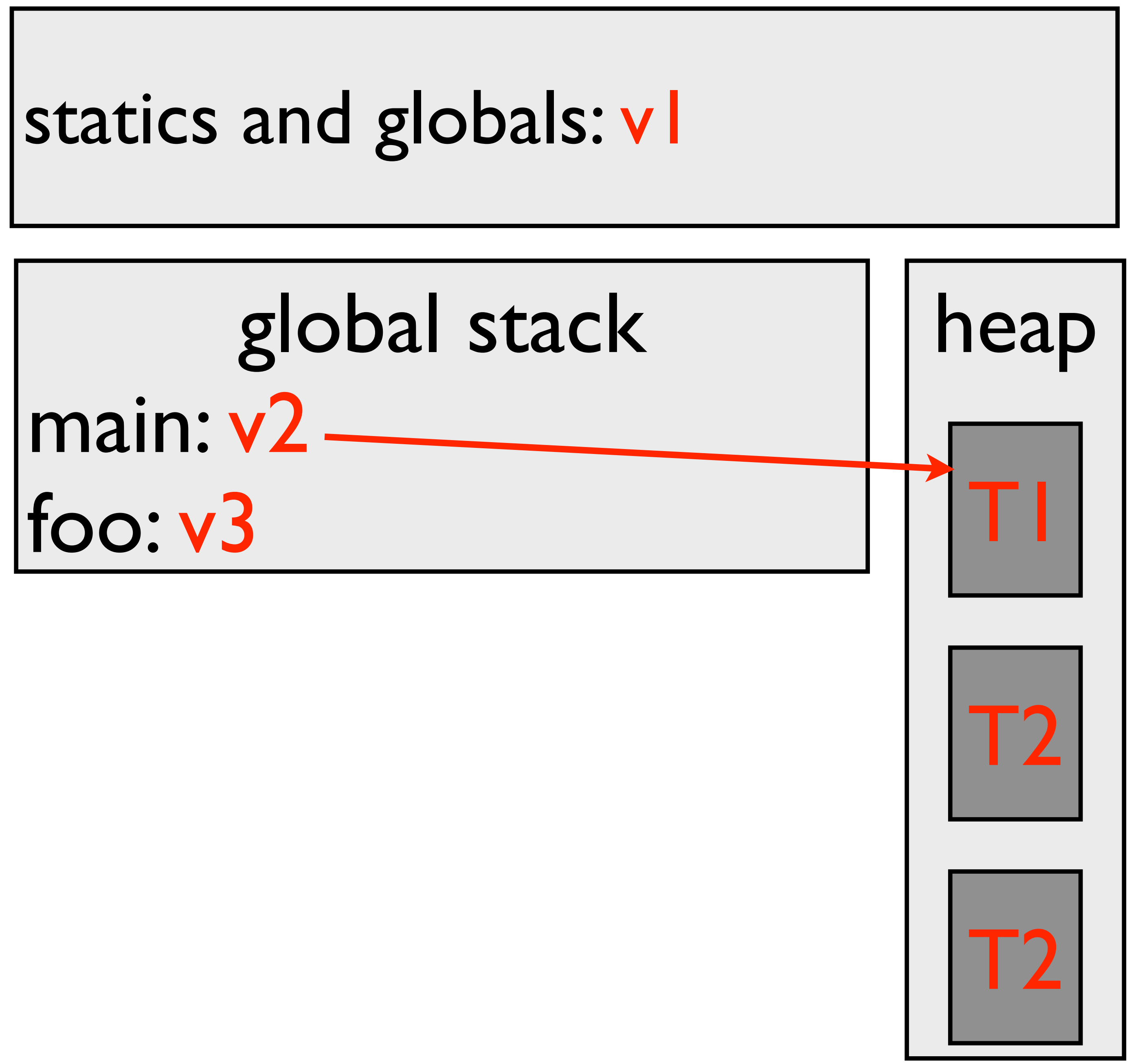


Context after *parallel for* finishes

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 private(v4,v5)

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

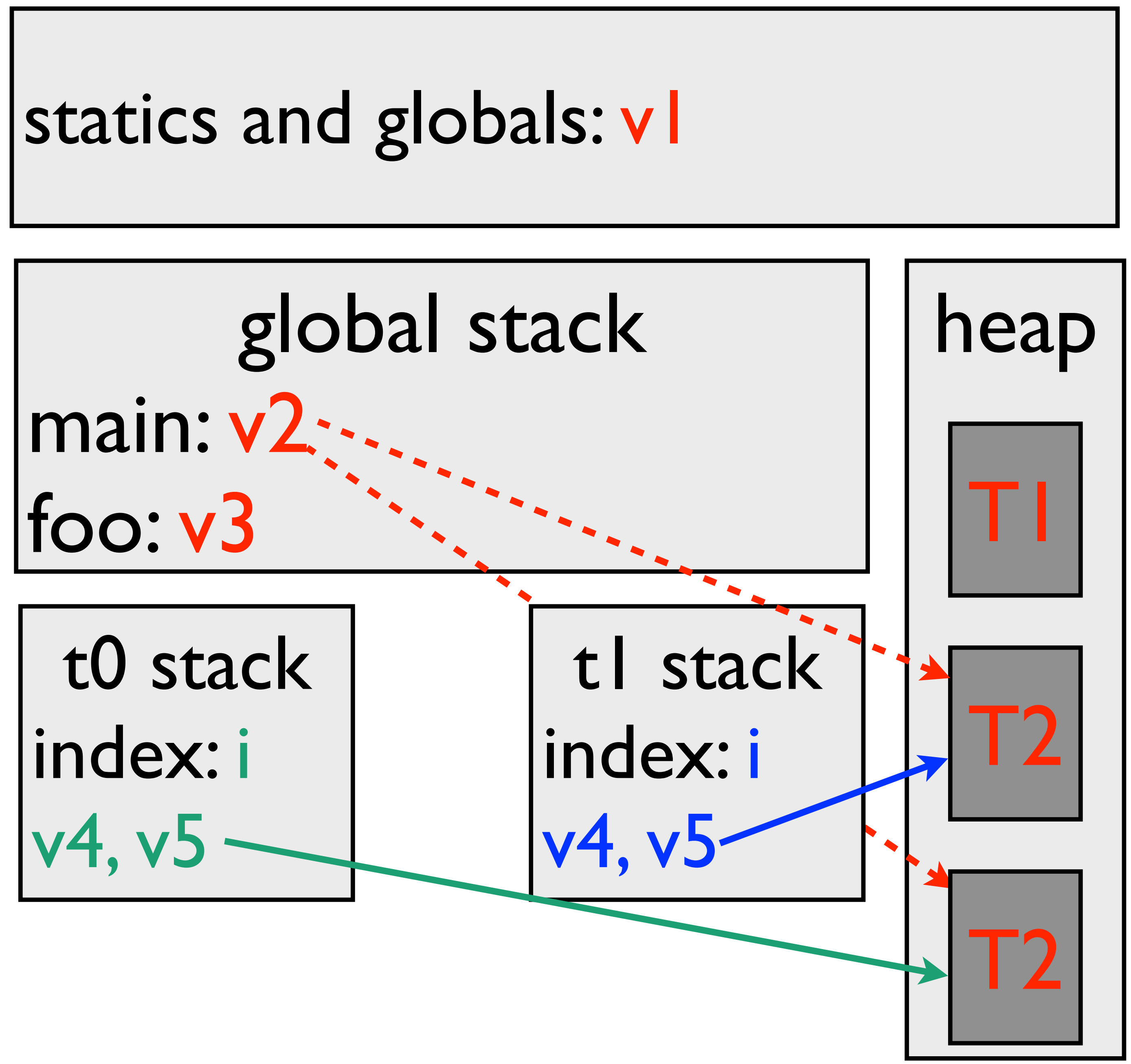


A slightly different example -- after each thread has run at least *1* iteration

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 private(v4,v5)

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

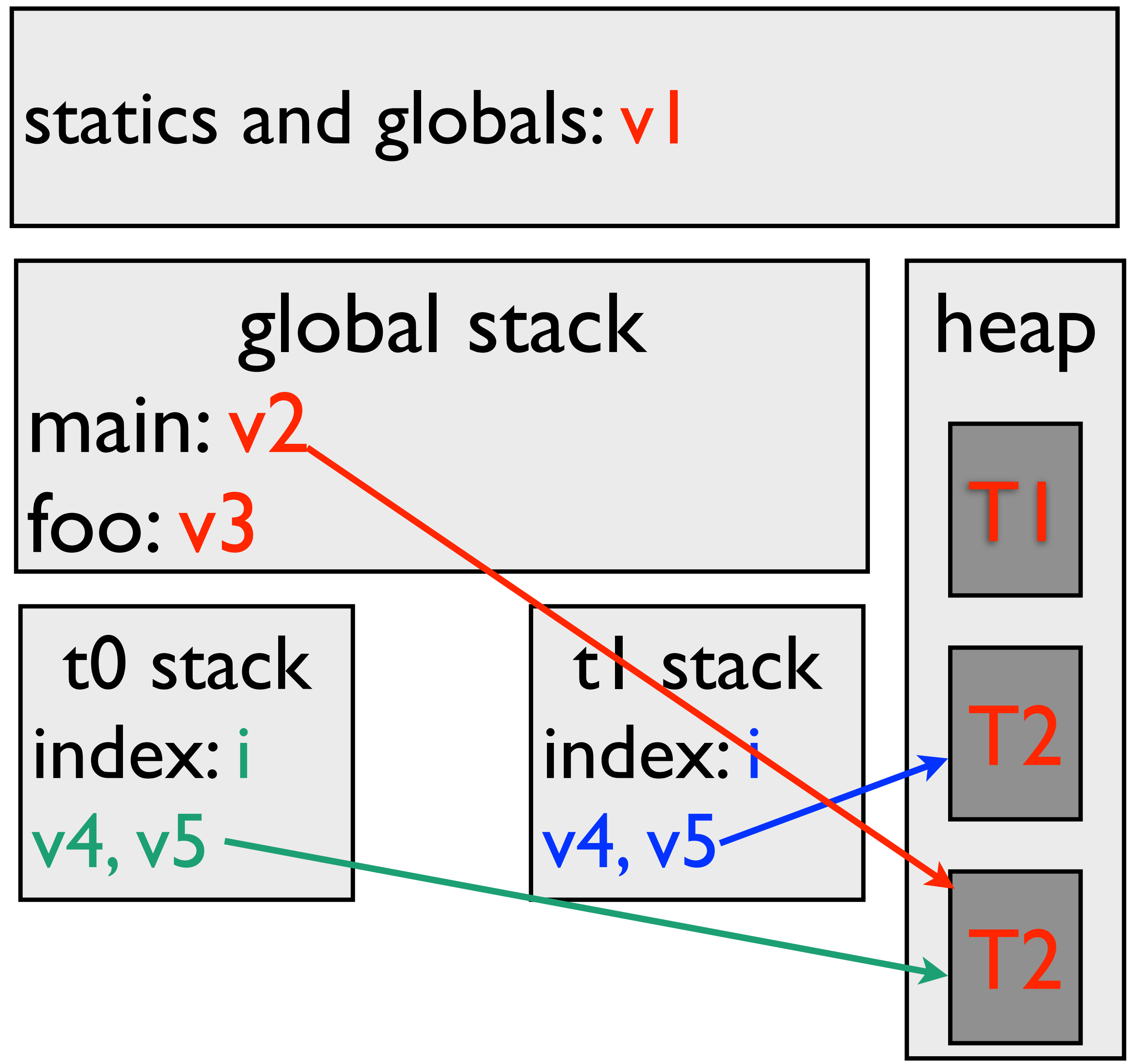


A slightly different example -- after each thread has run at least *1* iteration

v2 points to the T2 allocated by 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 private(v4,v5)

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

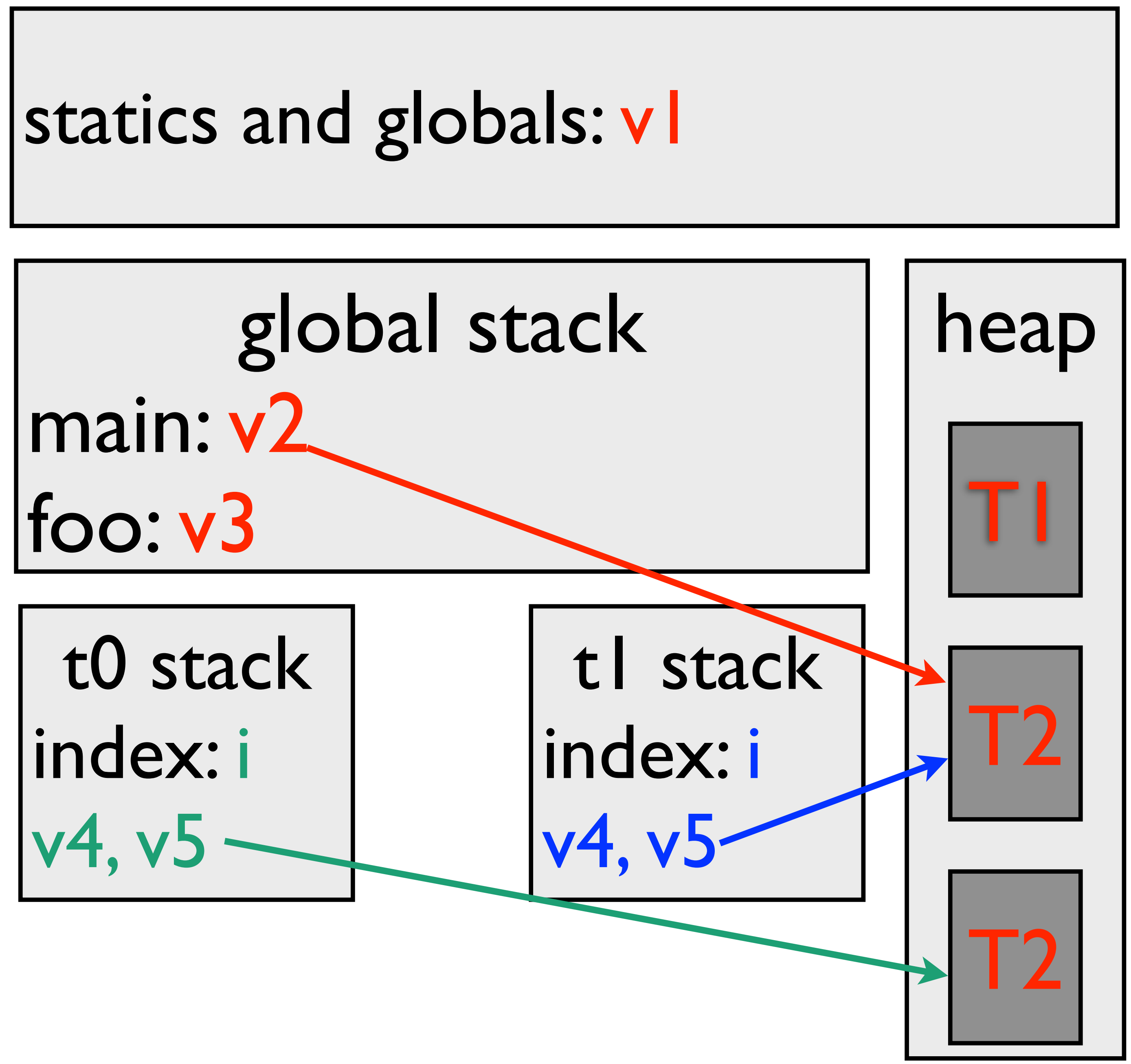


A slightly different example -- after each thread has run at least *1* iteration

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 private(v4,v5)

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

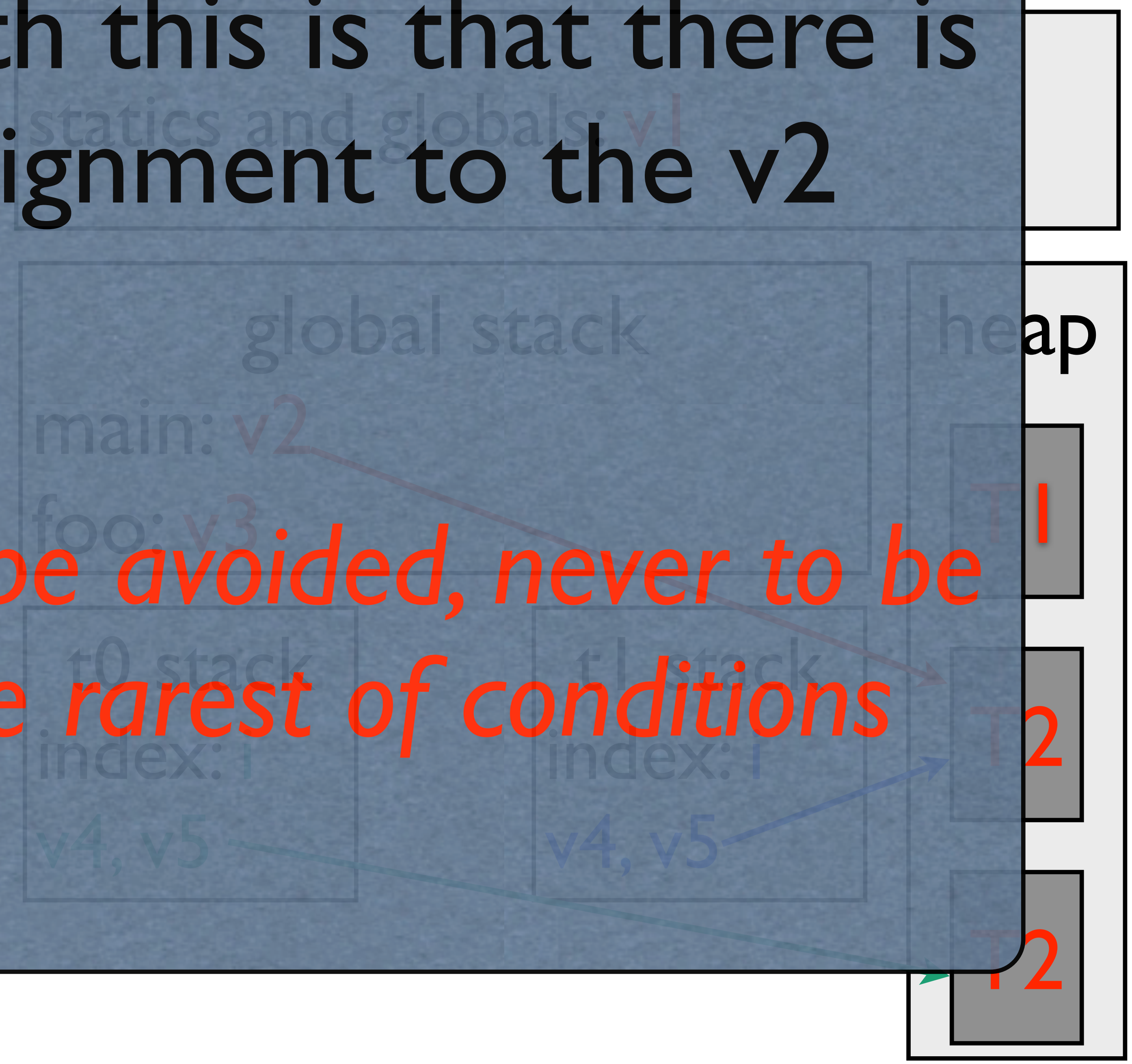


A slightly different example -- after each thread has run at least 1 iteration

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 private(v4,v5)

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



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

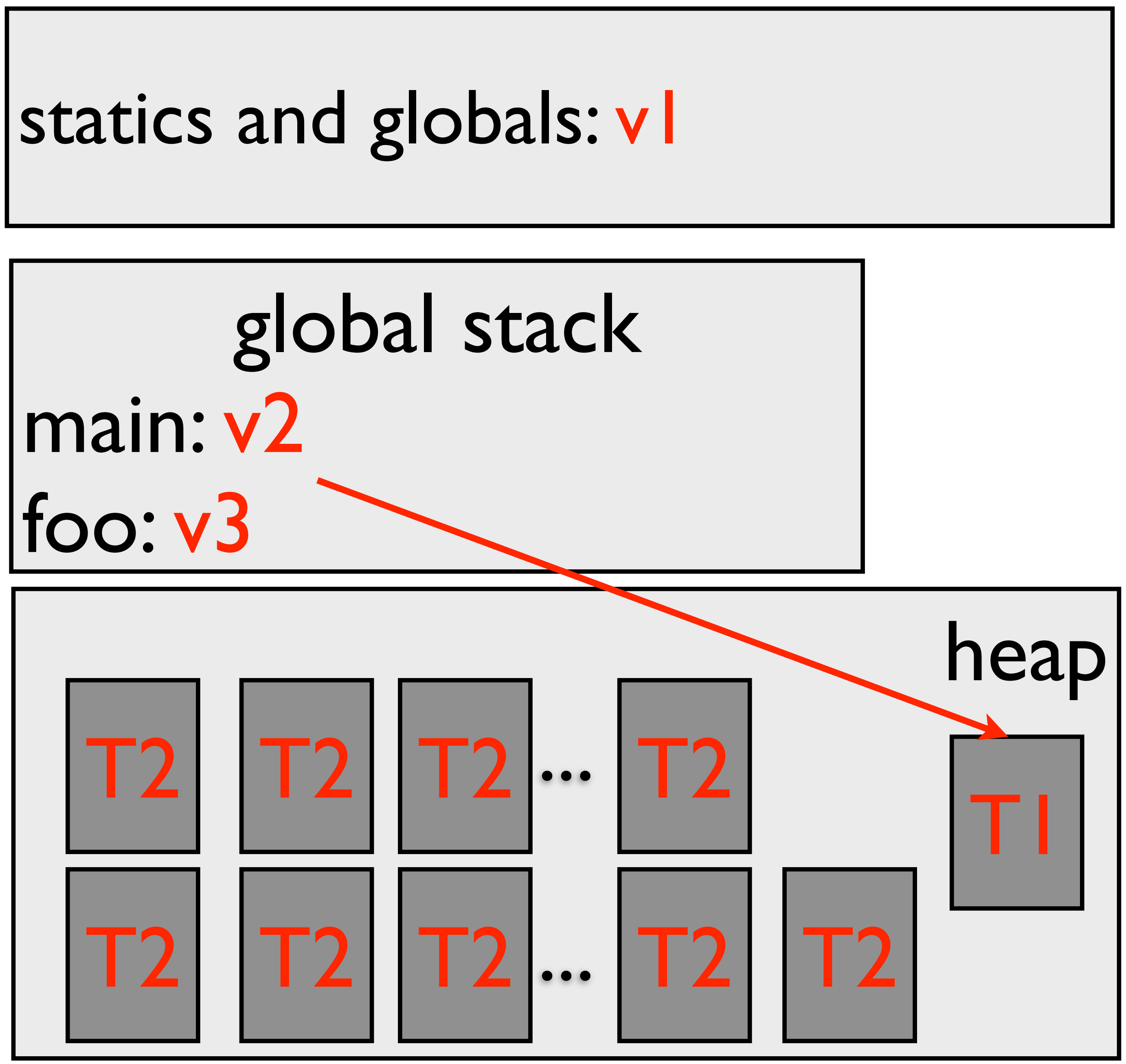
Races are evil, to be avoided, never to be done except in the rarest of conditions

Another problem with this code

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 private(v4,v5)

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



Querying the number of physical processors

- Can query the number of physical processors
    - returns the number of *cores* on a multicore machine
    - 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
  - 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

```
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

```
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*>)

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

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 private(tmp) 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? Depends on what assignment to *tmp* executes last

```
double tmp = 52;
#pragma omp parallel for private(tmp) 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 value written to a private variable in the *last iteration of the loop (n-1)* in the example below. Note that depending on the order that different threads execute their iterations, the last iteration of the loop executed by some thread may not be iteration n
- z and *tmp* will have the value of *tmp* 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;
```

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

What is wrong with this?

Second (and correct but slow) try:

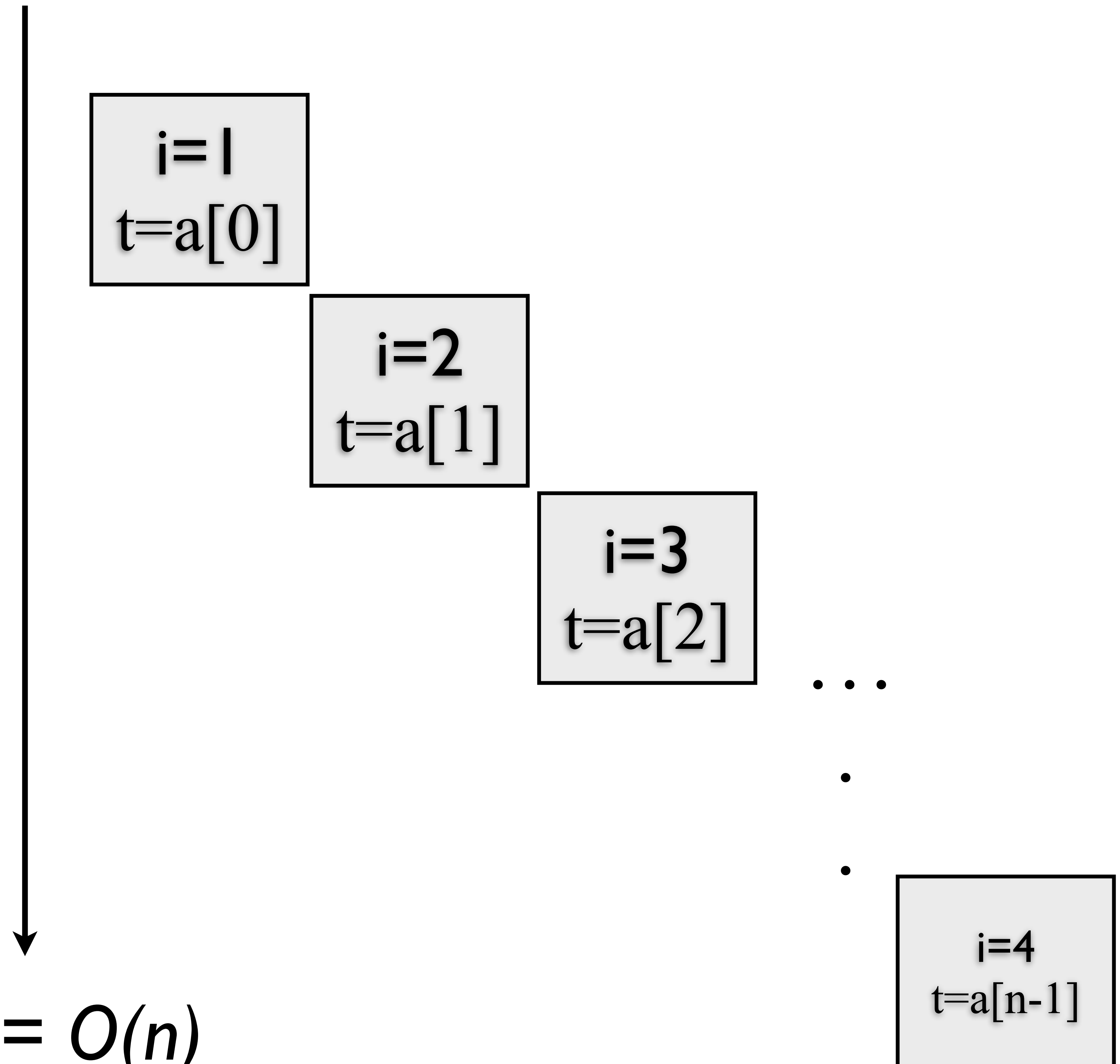
- use a *critical* section in the code
- executes the following (possible compound) statement atomically

```
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

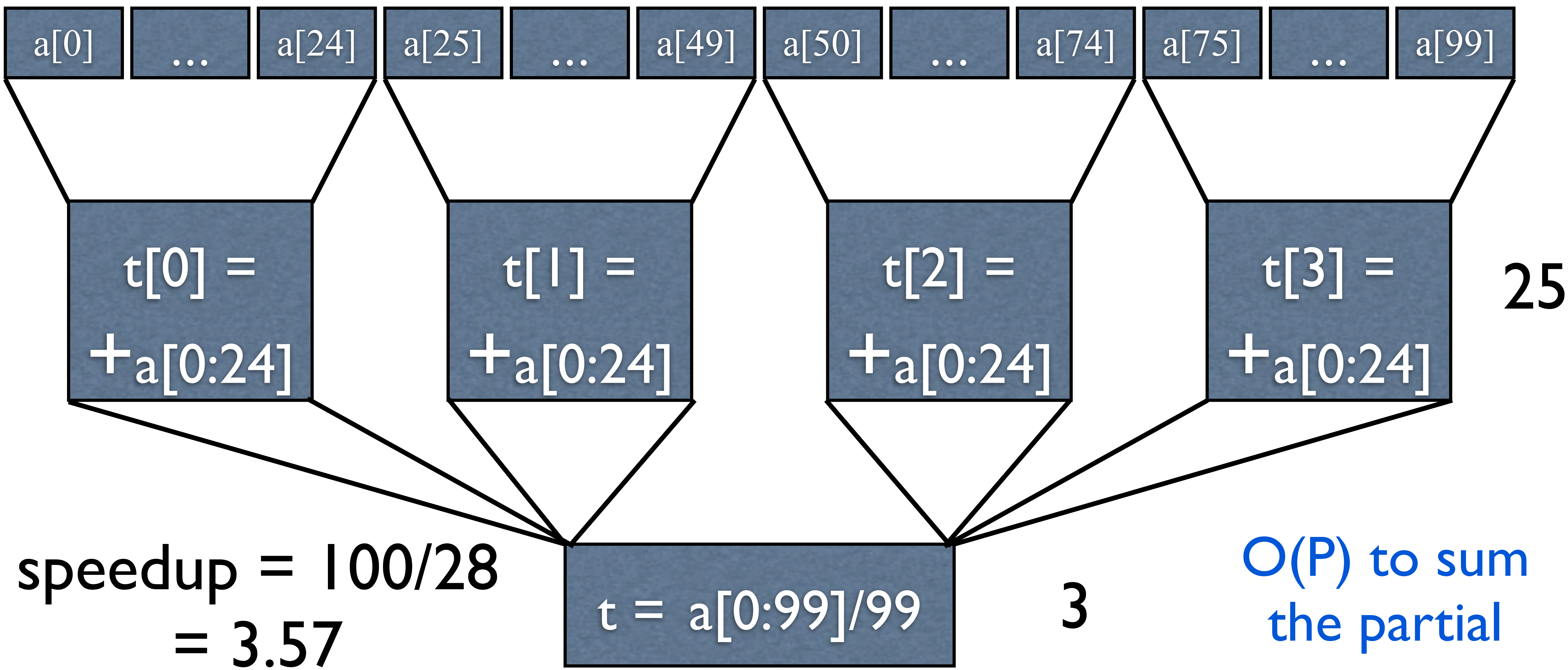
```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
   t = a[i];
}
t = t/n
```



This 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

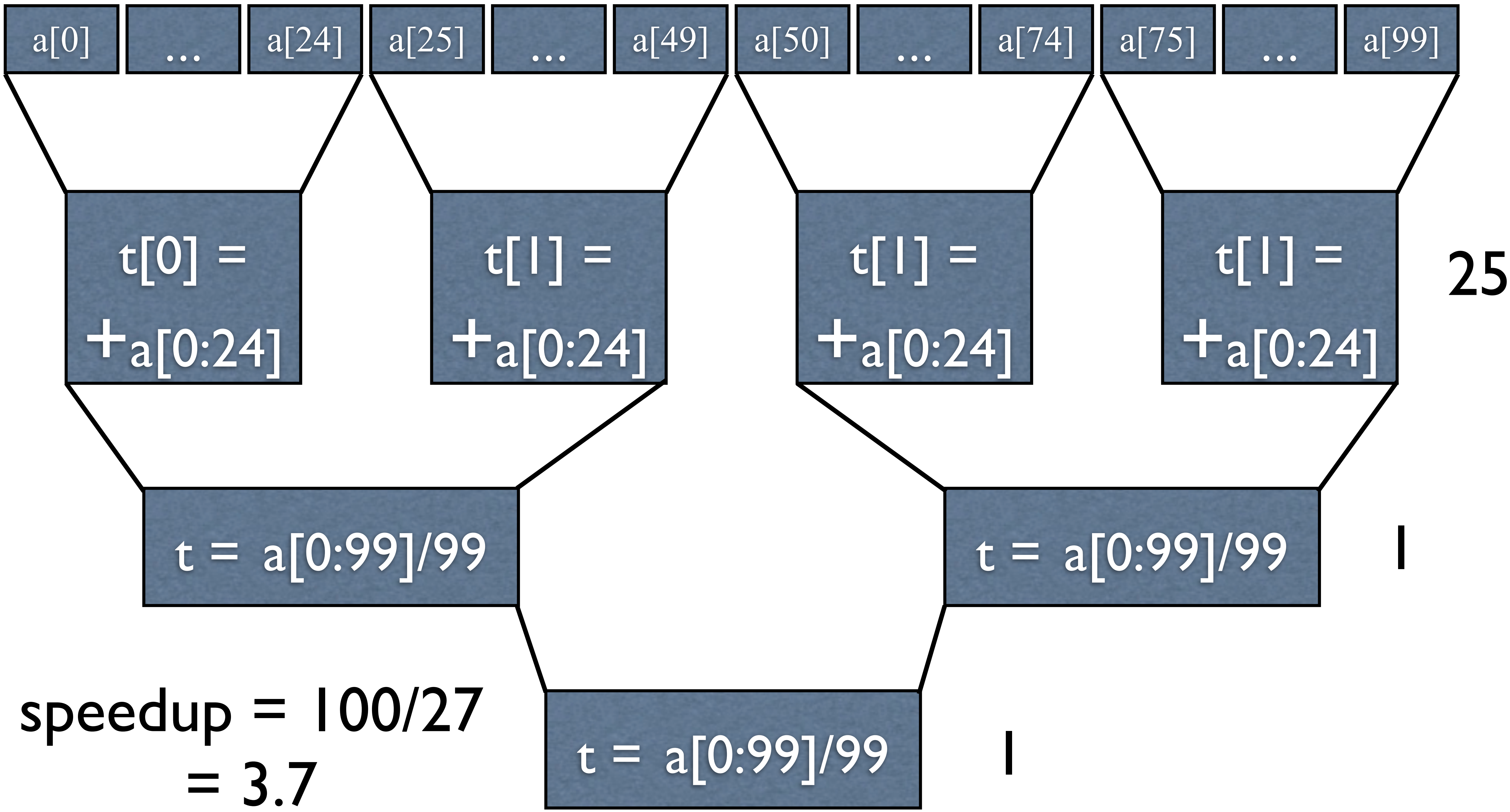
A parallel reduction



speedup = 100/28
= 3.57

O(P) to sum the partial sums

A better parallel reduction



speedup = 100/27
= 3.7