OpenMP

A programming model must provide a way of specifying

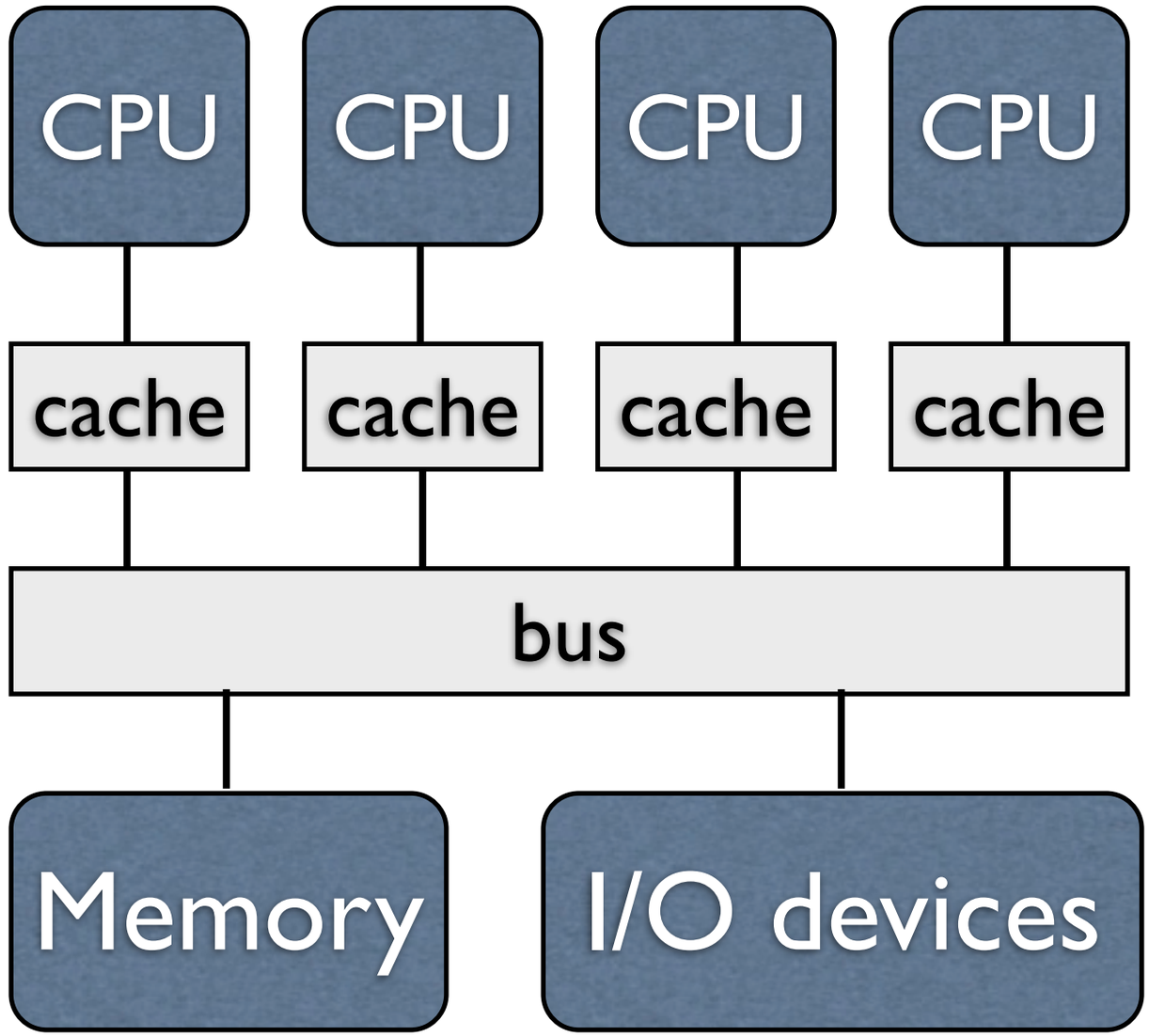
- what parts of the program execute in parallel with one another
- how the work is distributed across different cores
- the order that reads and writes to memory will take place
- that a sequence of accesses to a variable will occur *atomically* or without interference from other threads.
- **And,**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

What 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

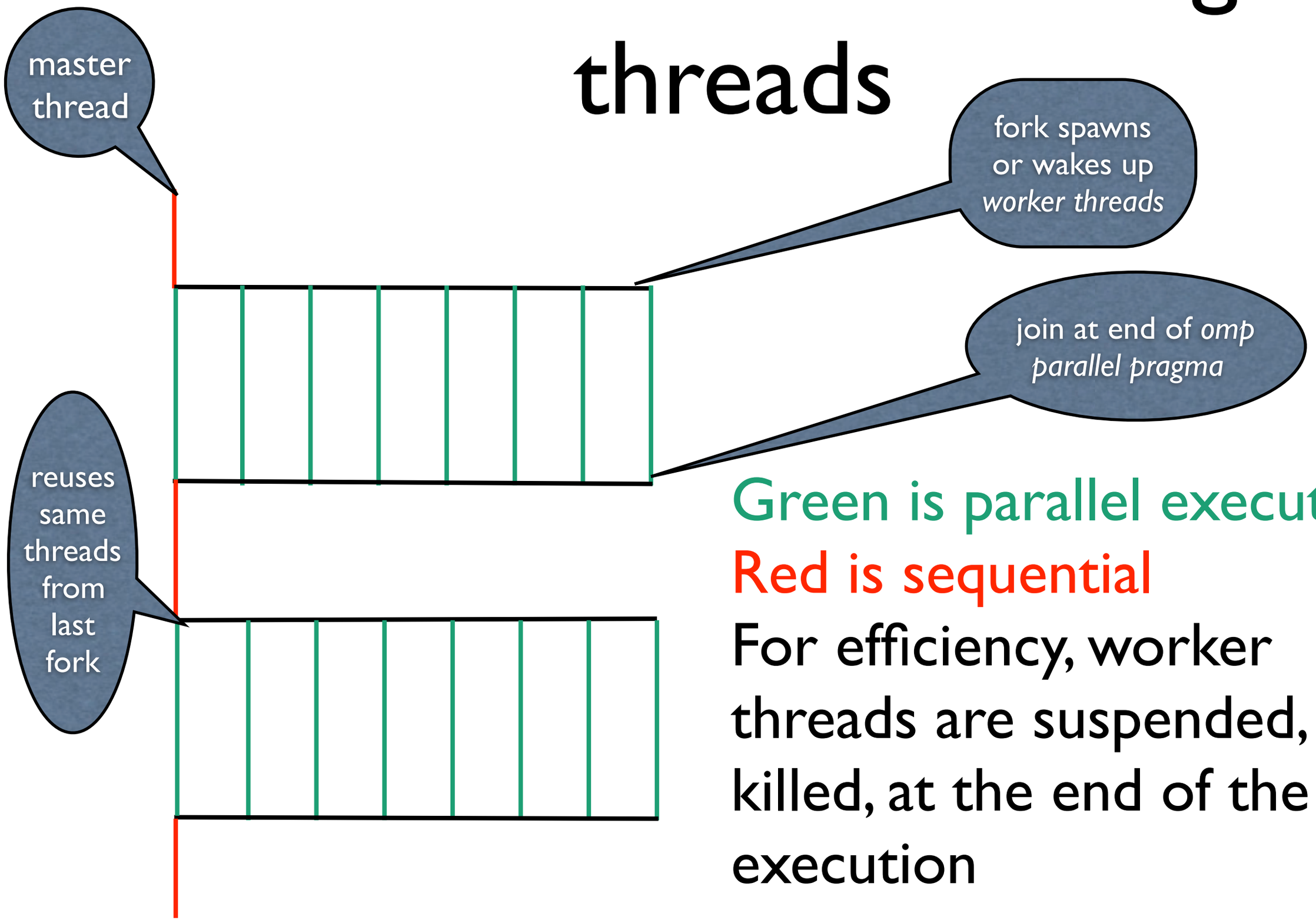
Uniform memory access shared memory machine is assumed



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

An aside -- threads and processes

- Threads and processes are typically operating system entities and concepts
- A *process* has its own address space and owns a typically *virtualized* copy of the machine when executing
  - processes may own one or more threads
- A *thread* shares its address space with it's owning process and all other threads owned by the same process
  - each thread has its own copy of registers
  - local variables can be created that are accessible only by the thread
  - threads are the fundamental building block of parallel shared memory programs

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 (ignoring roundoff error and stability issues)
- #pragma omp <*rest of the pragma*> is the general form of a pragma

pragma to create a parallel for

OpenMP programmers use the *parallel for* pragma to tell the compiler a *for* 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 - these restrictions basically prevent* while *loops represented as a* for *loop which are allowed by C/C++/Fortran*

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

potentially shared/
must be 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;
        T2 *v5 = malloc(sizeof(T2));
         v3 = (int) *v5;
}}
```

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

Because v4 and v5 are declared inside the parallel loop they are automatically thread 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 shared(v4,v5)
        for (int i=0; i < n; i++) {
        int v4;
        T2 *v5 = malloc(sizeof(T2));
         v3 = (int) *v5;
}}
```

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

Because v4 and v5 are declared inside the parallel loop they are automatically thread private.

If you want these variables to be shared across all threads used the shared clause.

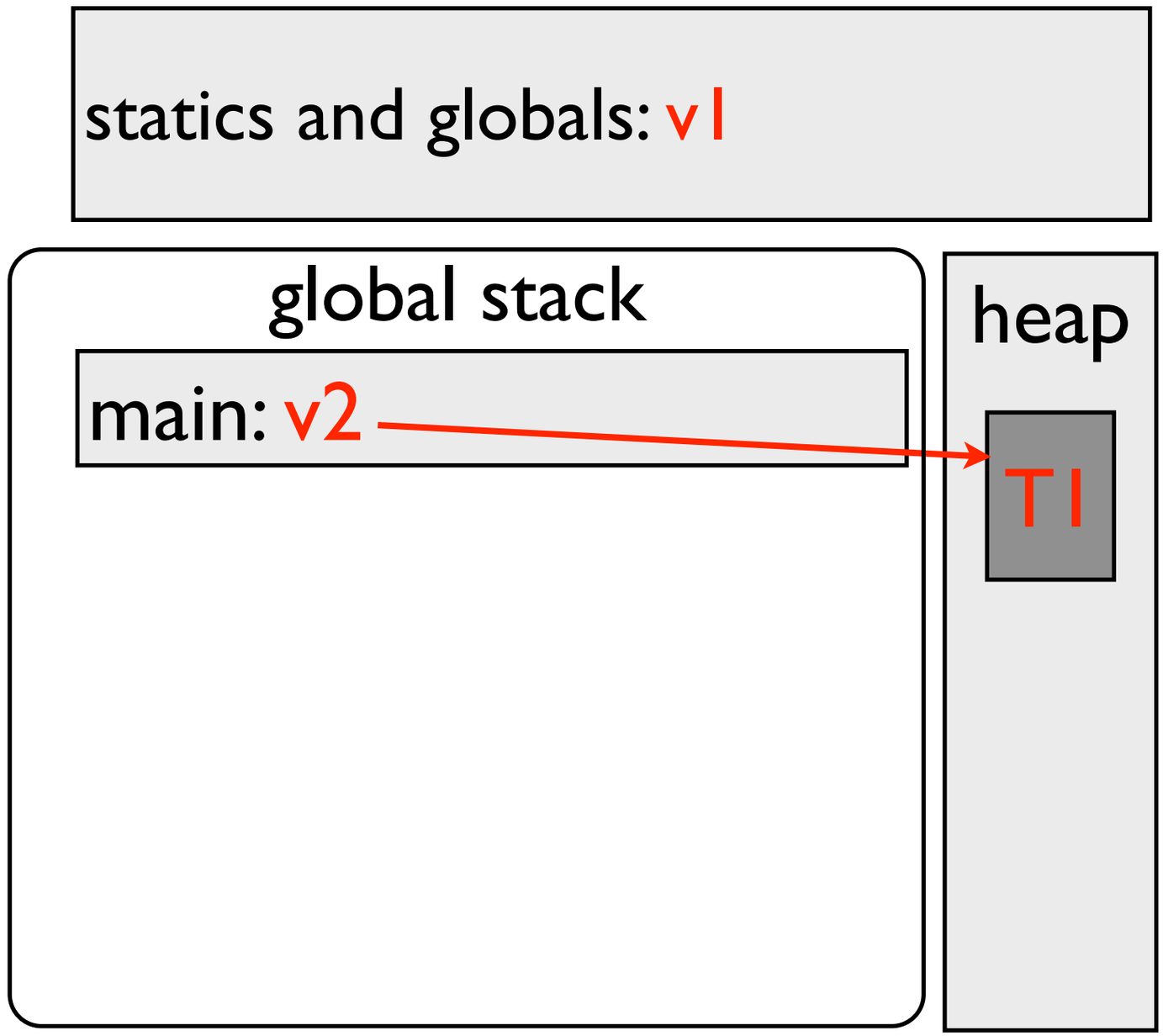
Context before call to f1

Storage, assuming two threads

red is potentially shared, , green is private to thread 0,

blue is private to thread i

```
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));
}}
```



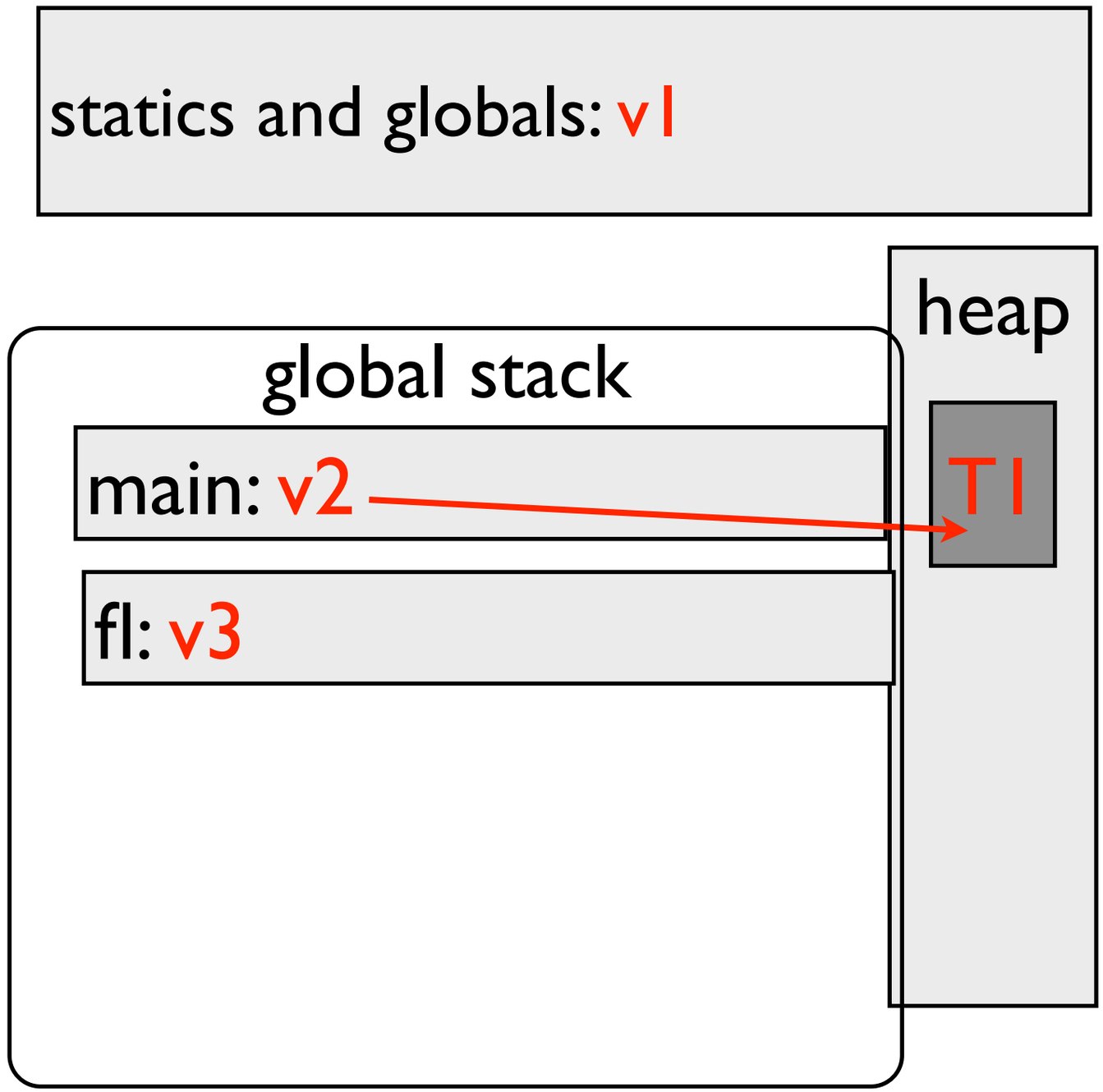
Context right after call to f1

Storage, assuming two threads

red is potentially shared, , green is private to thread 0,

blue is private to thread i

```
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));
}}
```



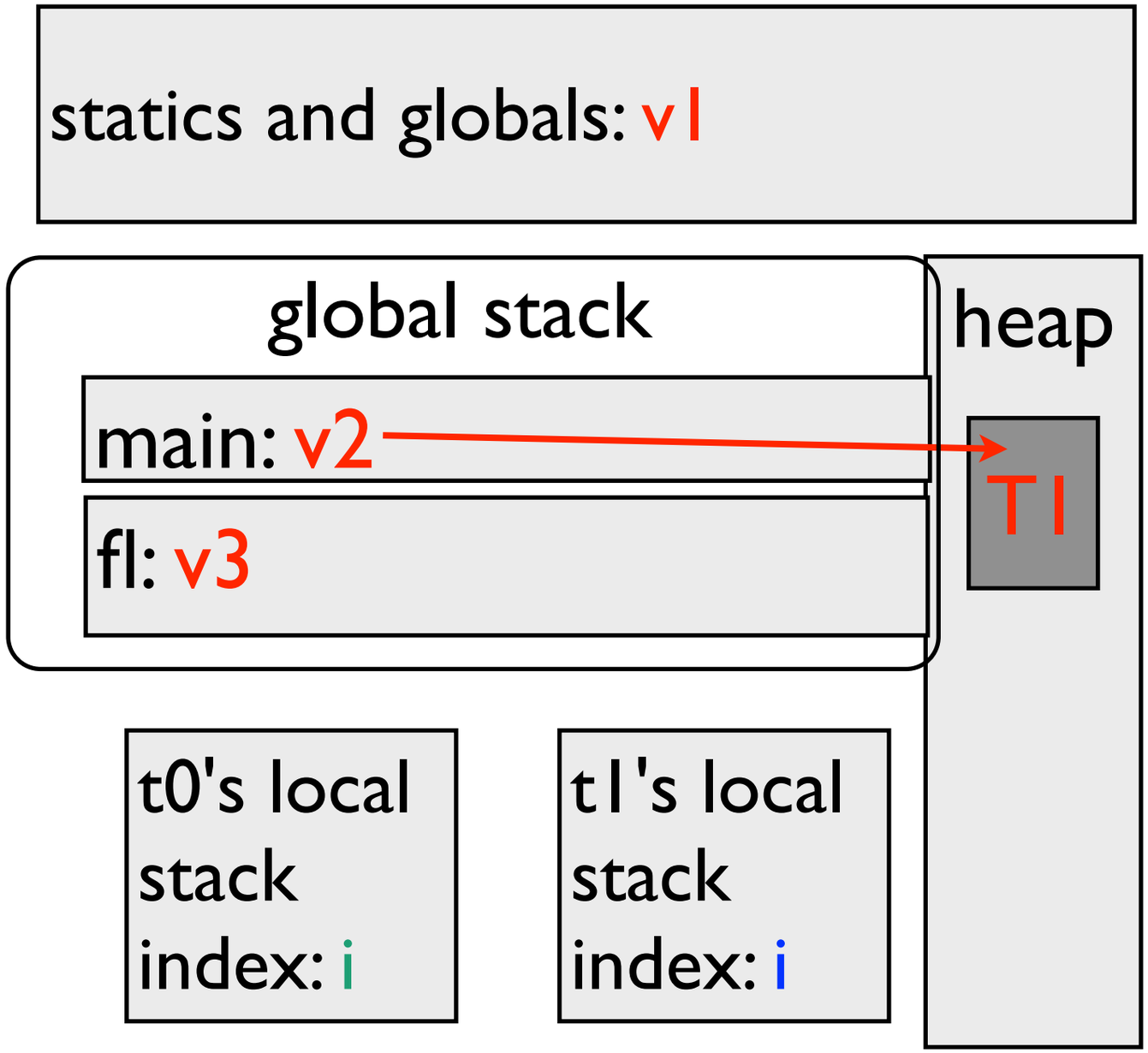
Context at start of parallel for

Storage, assuming two threads

red is potentially shared, , green is private to thread 0,

blue is private to thread i

```
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));
}}
```



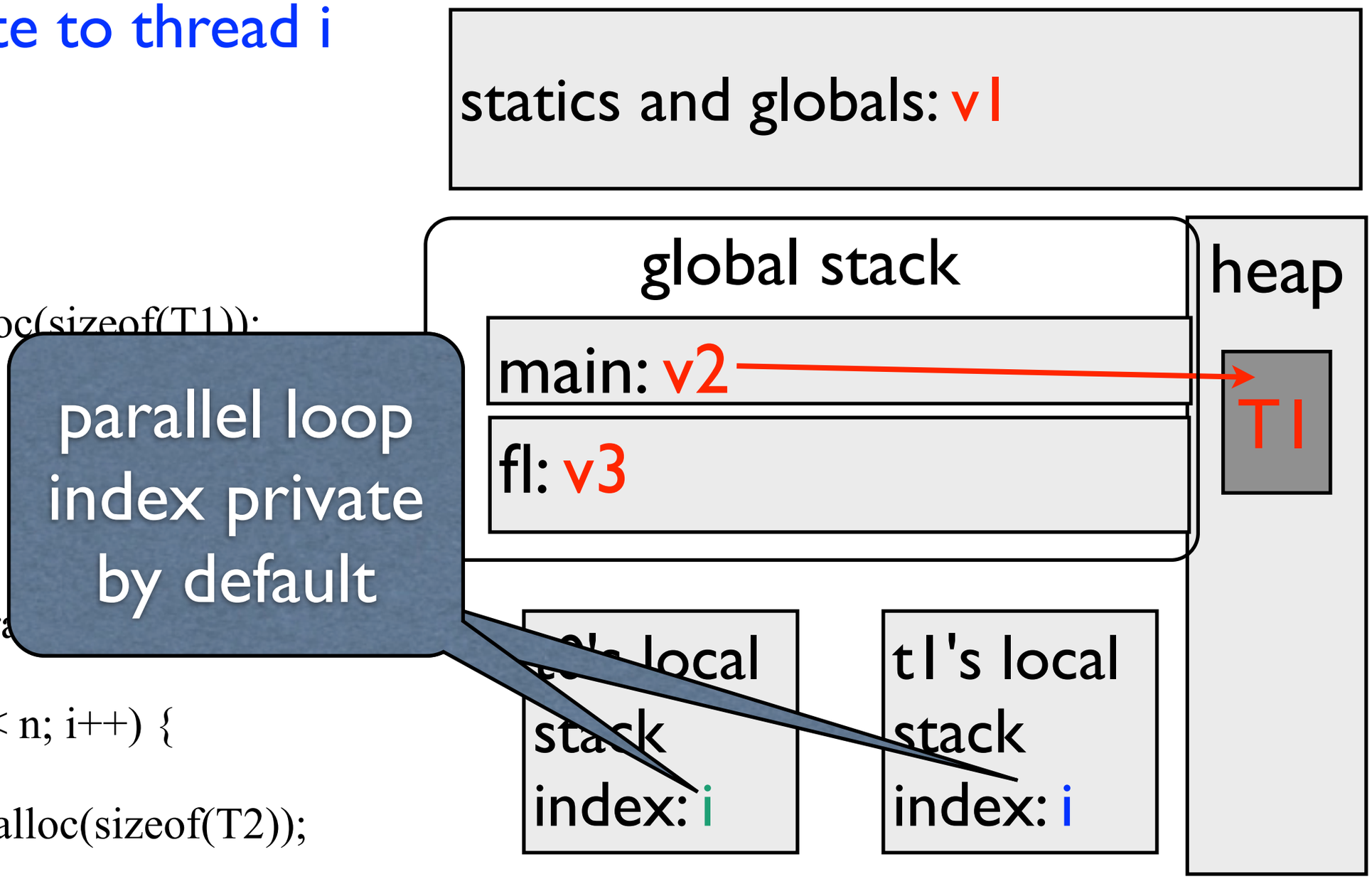
Context at start of parallel for

Storage, assuming two threads

red is potentially shared, , green is private to thread 0,

blue is private to thread i

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

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



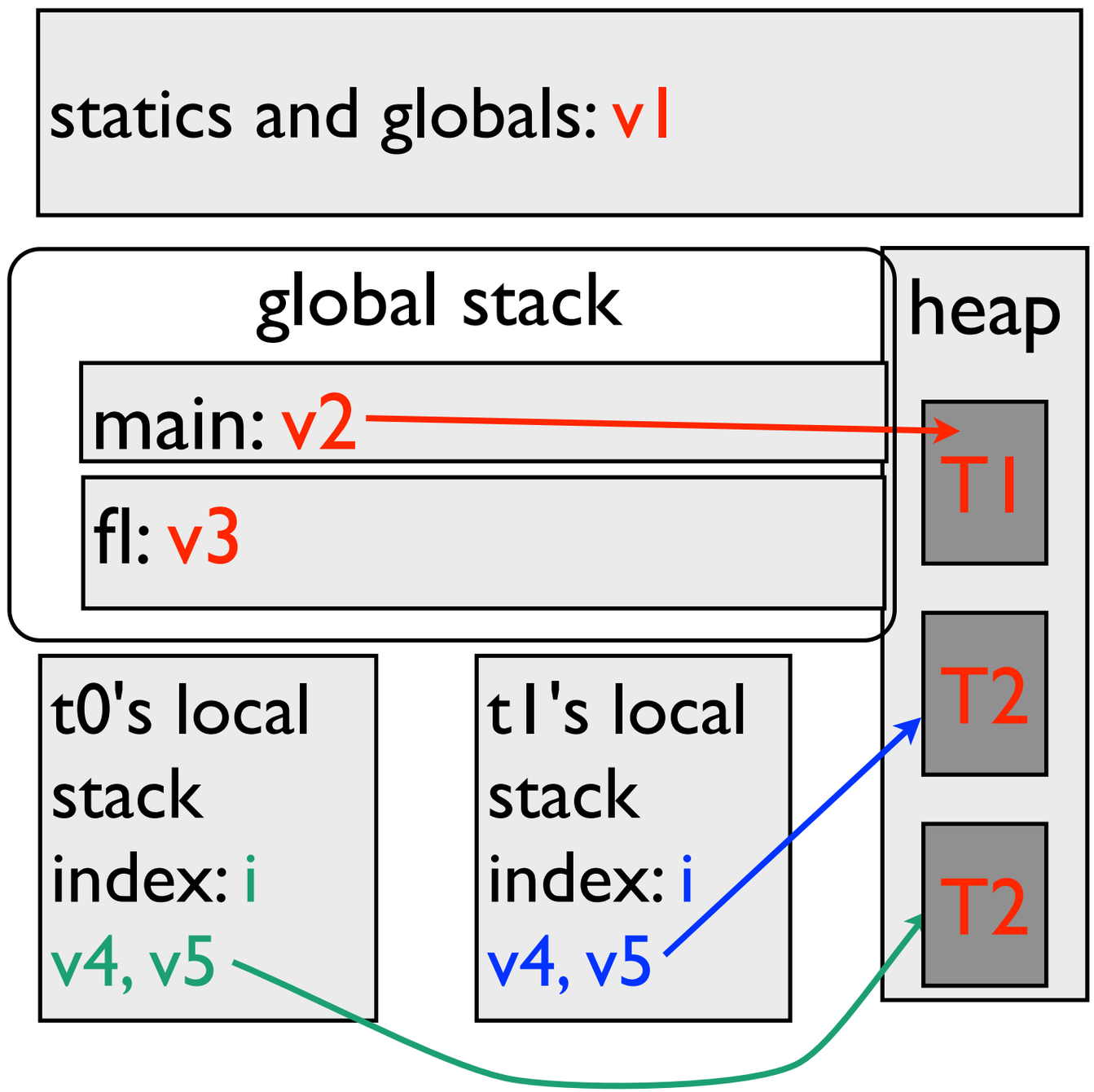
Context after first iteration of the *parallel for*

Storage, assuming two threads

red is potentially shared, , green is private to thread 0,

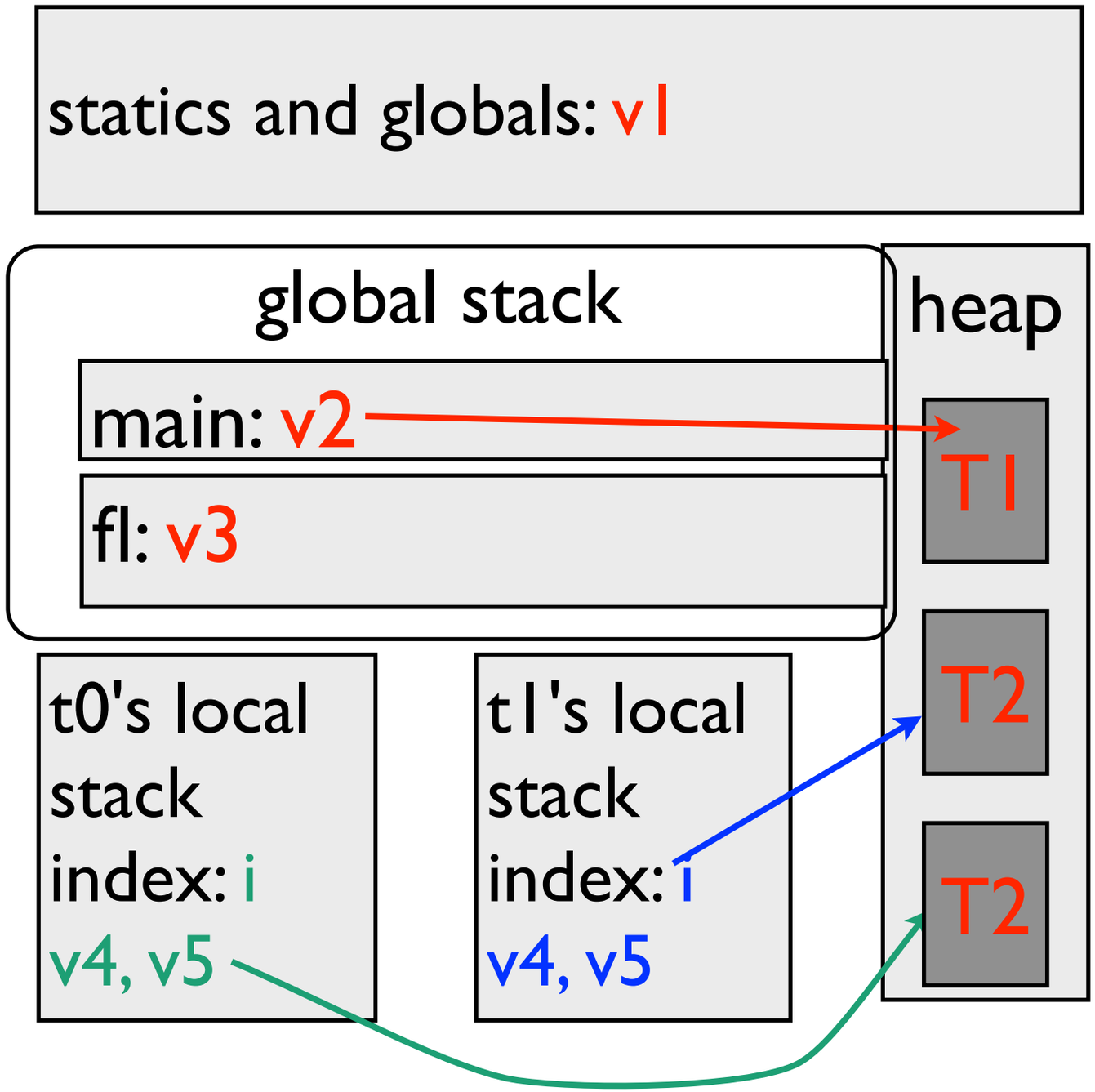
blue is private to thread i

```
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));
}}
```



Is malloc thread safe? It usually is, but don't count on it and check the documentation.

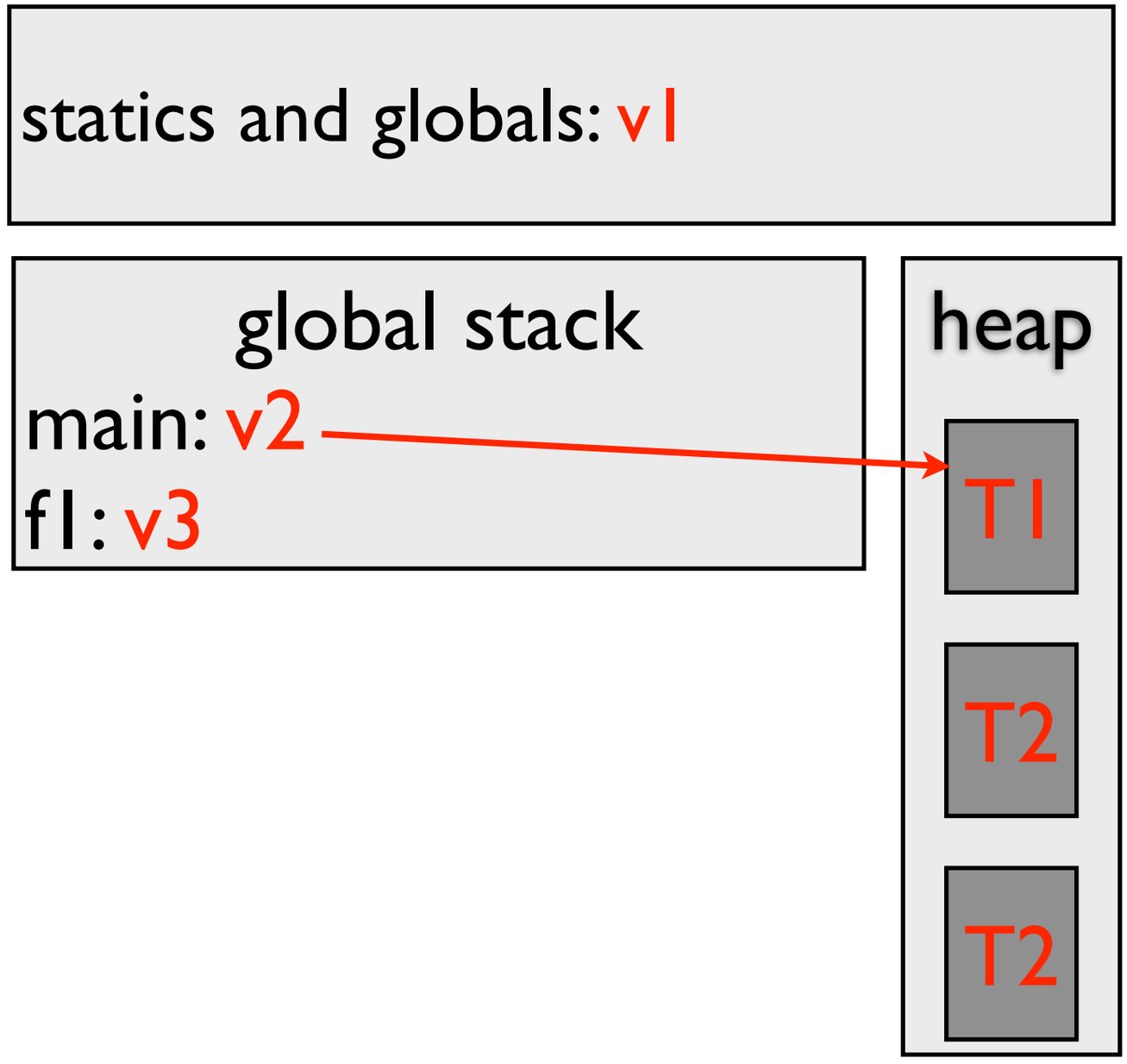
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for

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



Context after *parallel for* finishes

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

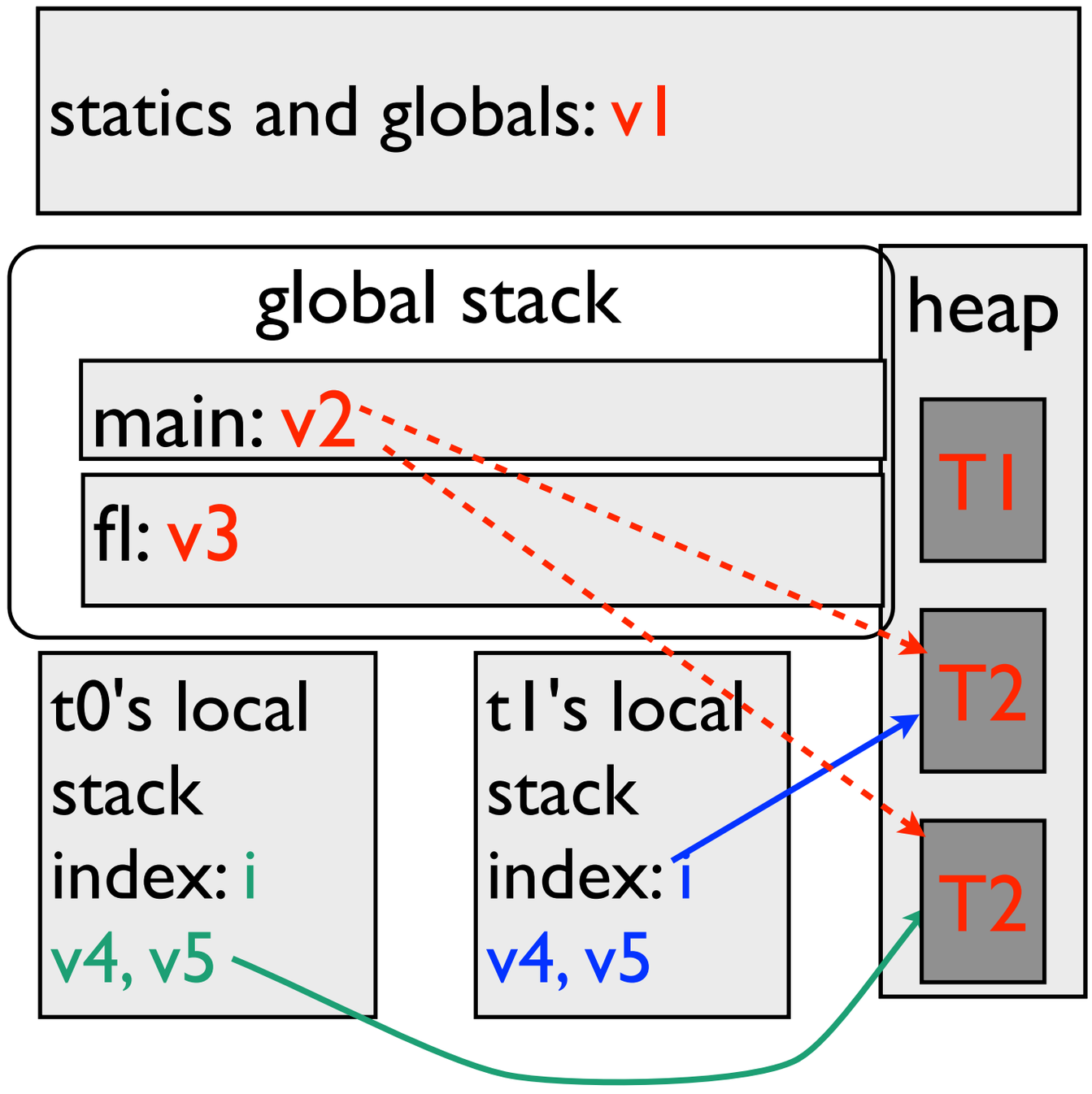
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;

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



A slightly different example -- after two iterations executes (*i=0 may not execute first*)

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

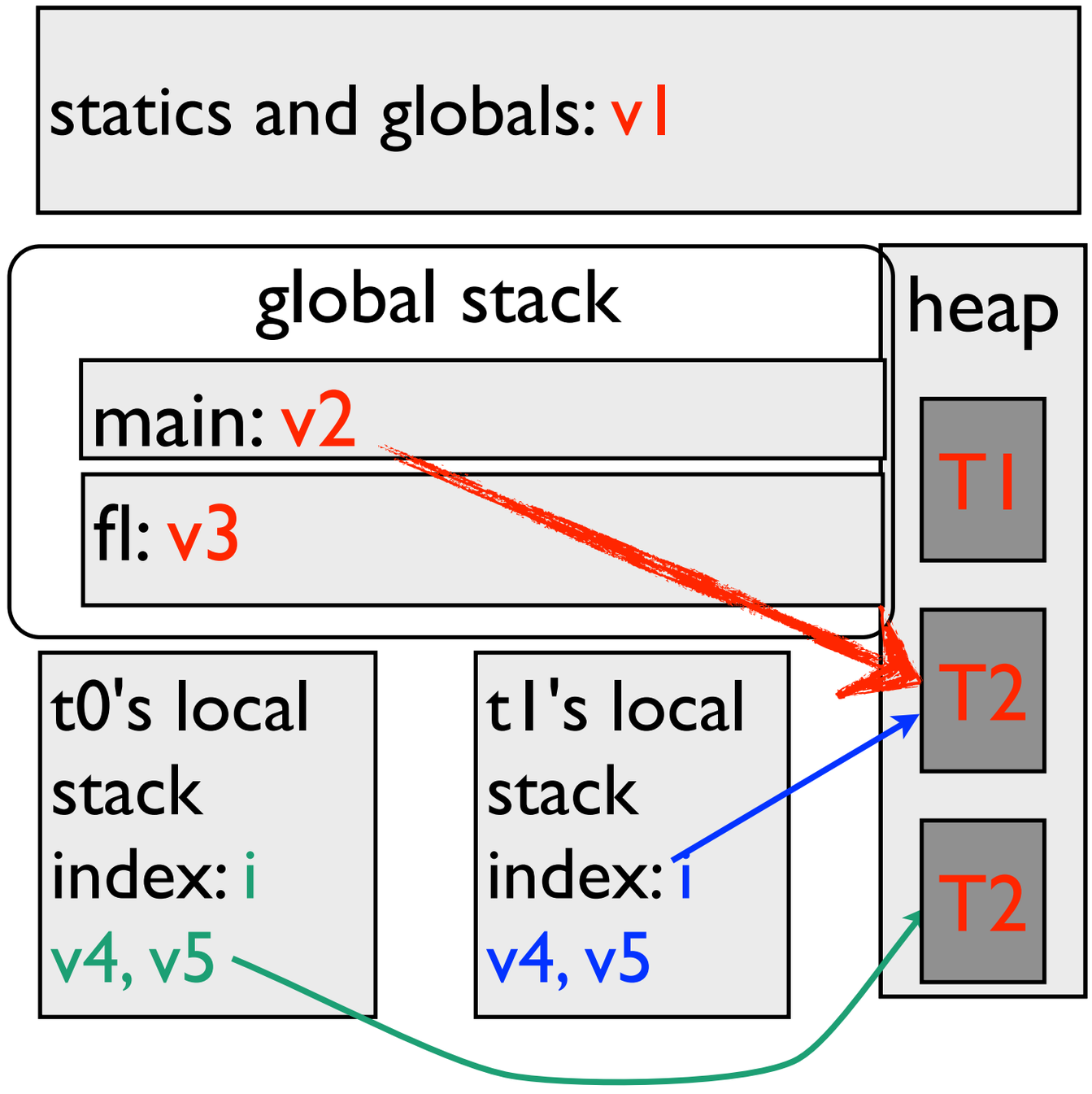
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for

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



If an iteration on t0 executes first

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

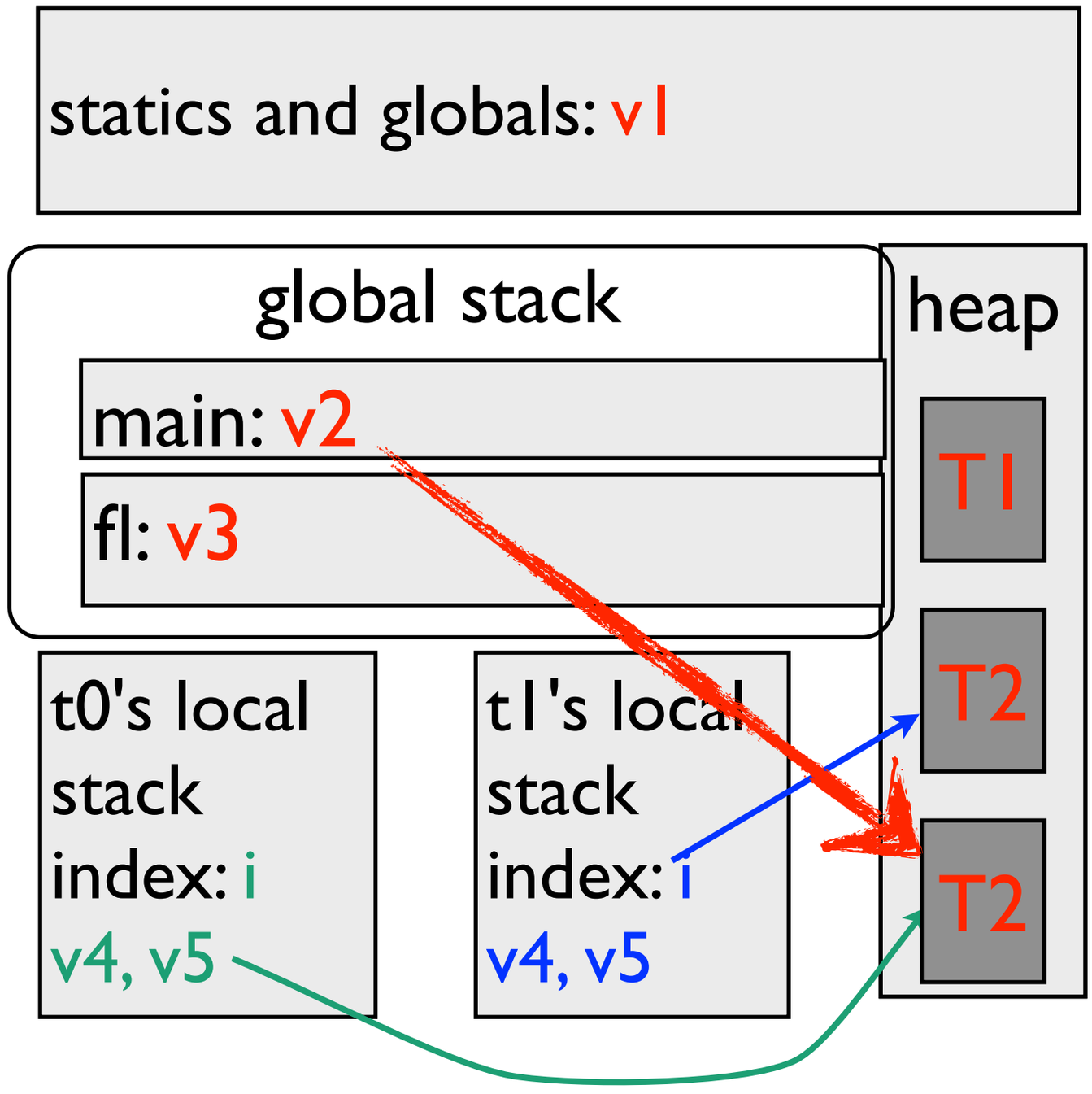
```
int v1;
...
main( ) {
    T1 *v2 = malloc(sizeof(T1));
    ...
    f1( );
}
void f1( ) {
    int v3;
#pragma omp parallel for

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



If an iteration on t1 executes first

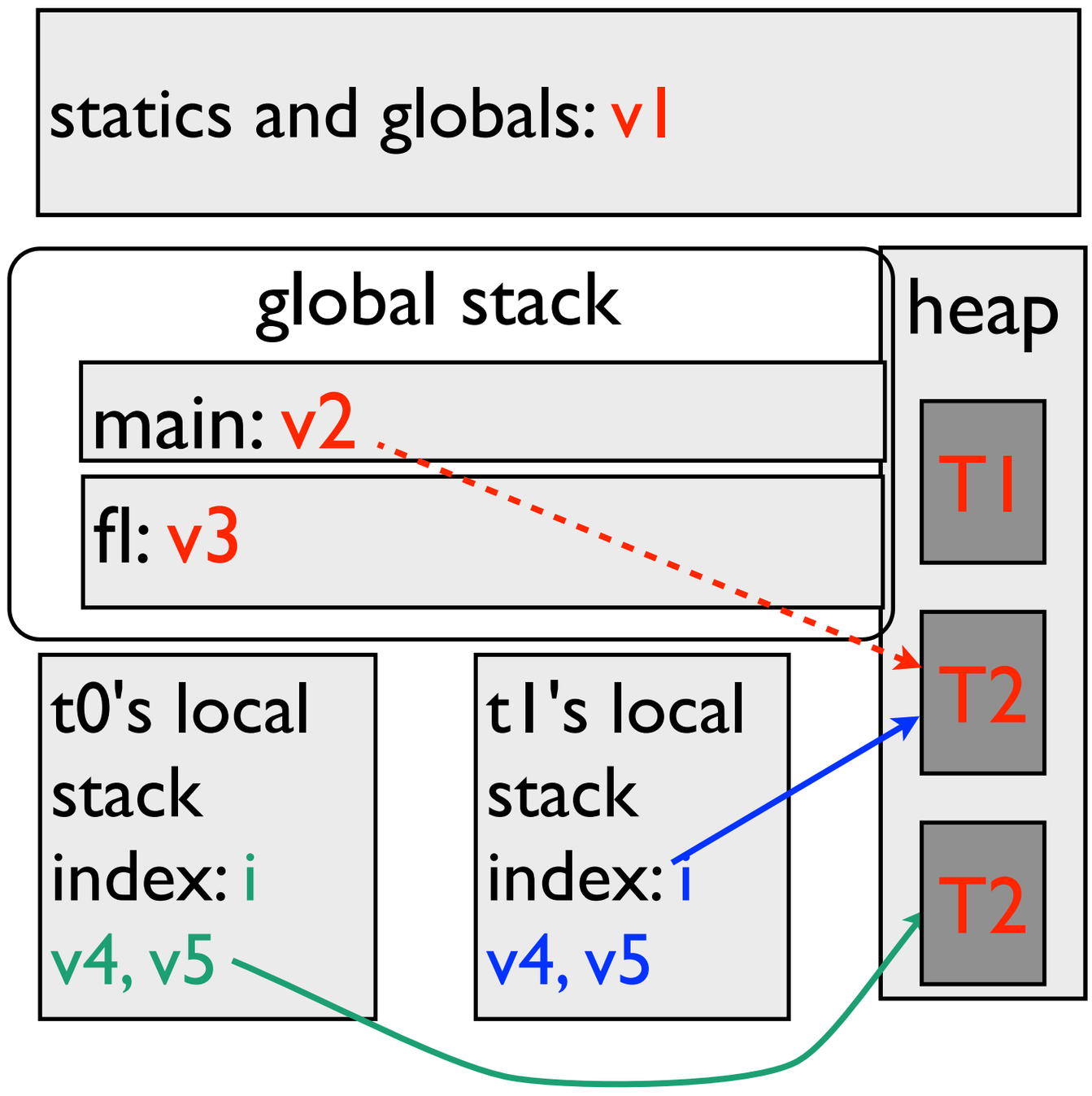
v2 points to one of the T2 objects that was allocated.
Which one? It depends.

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

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



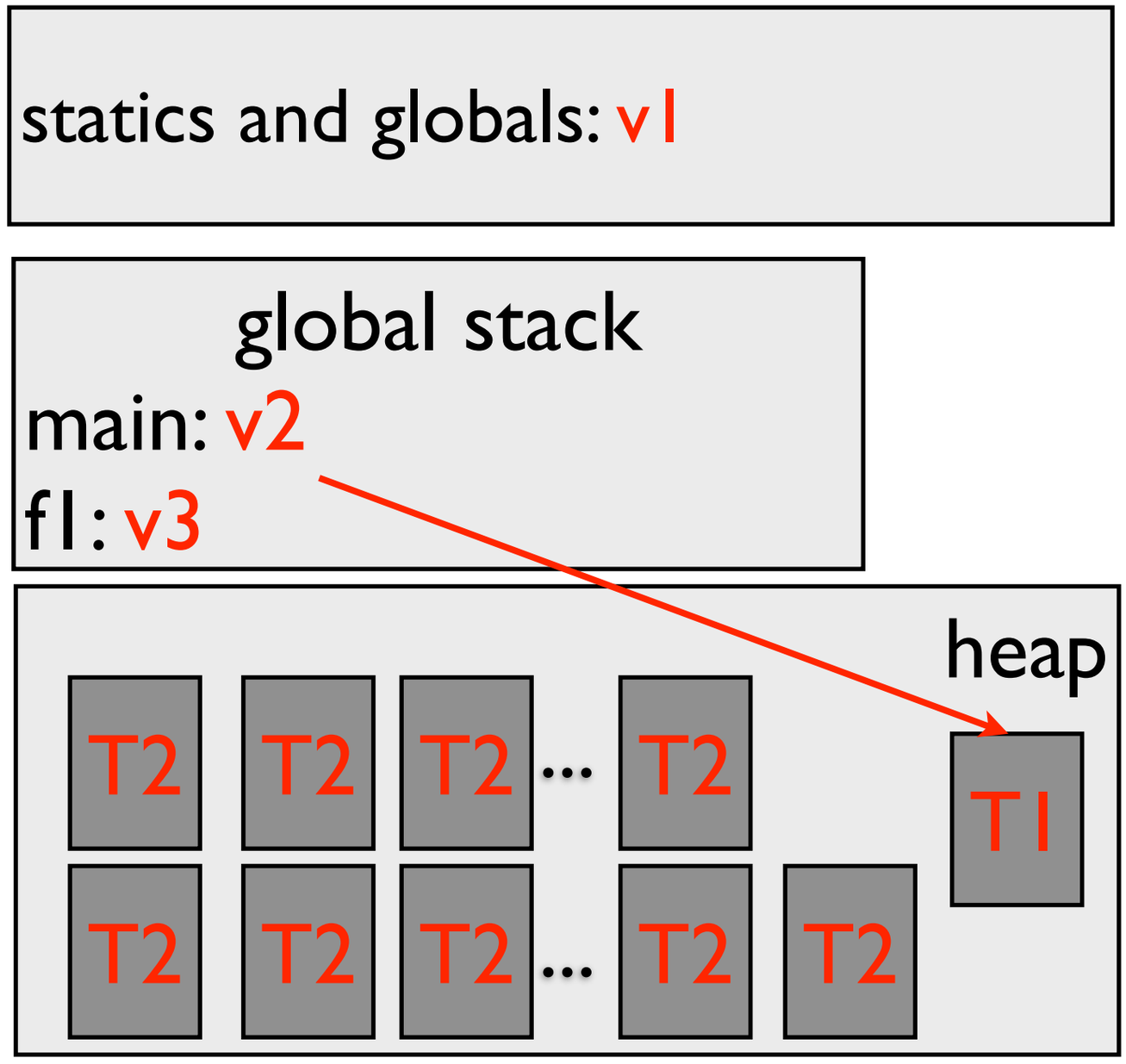
When the program outcome depends on the order unsynchronized data is accessed we have a *race*

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

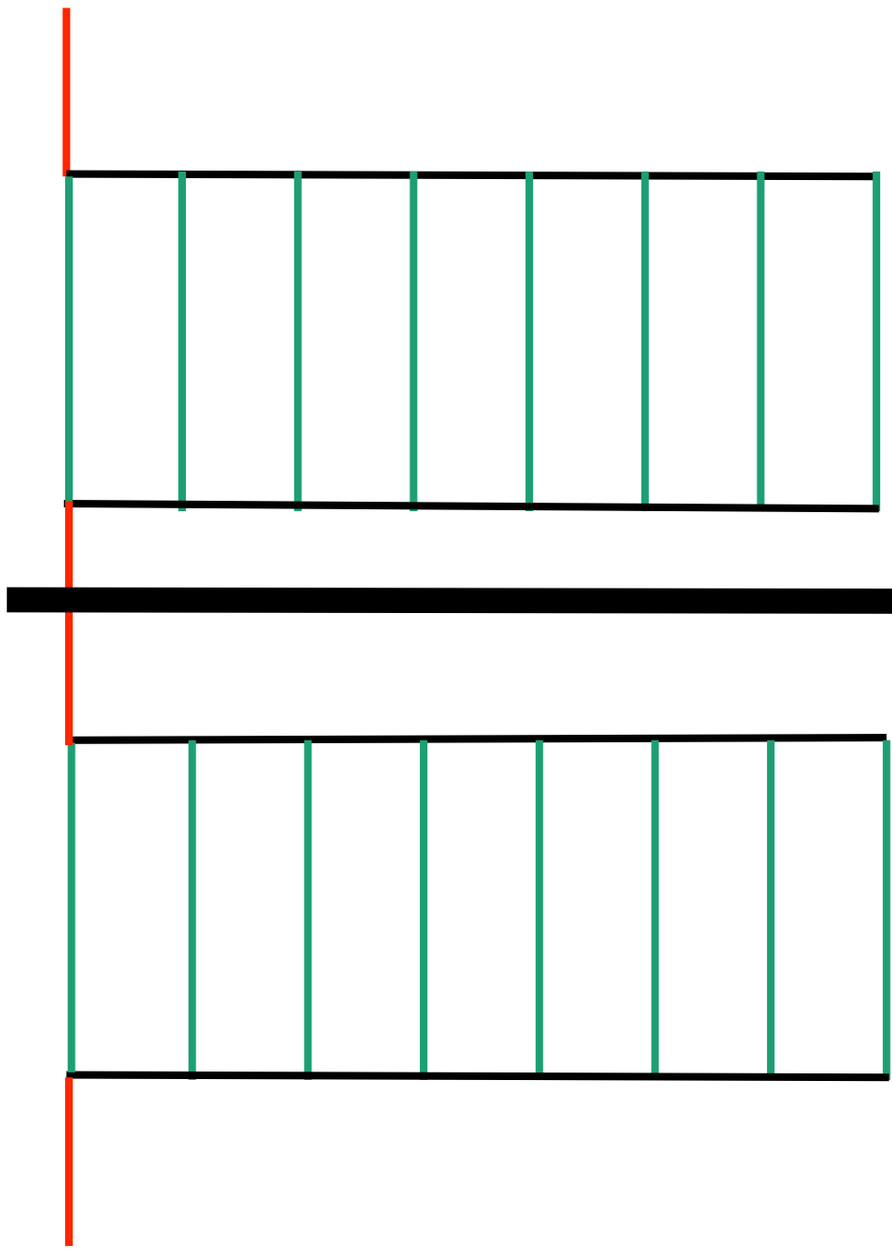


Another problem with this code -- there is a memory leak

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

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



By default, every parallel construct is followed by a *barrier*

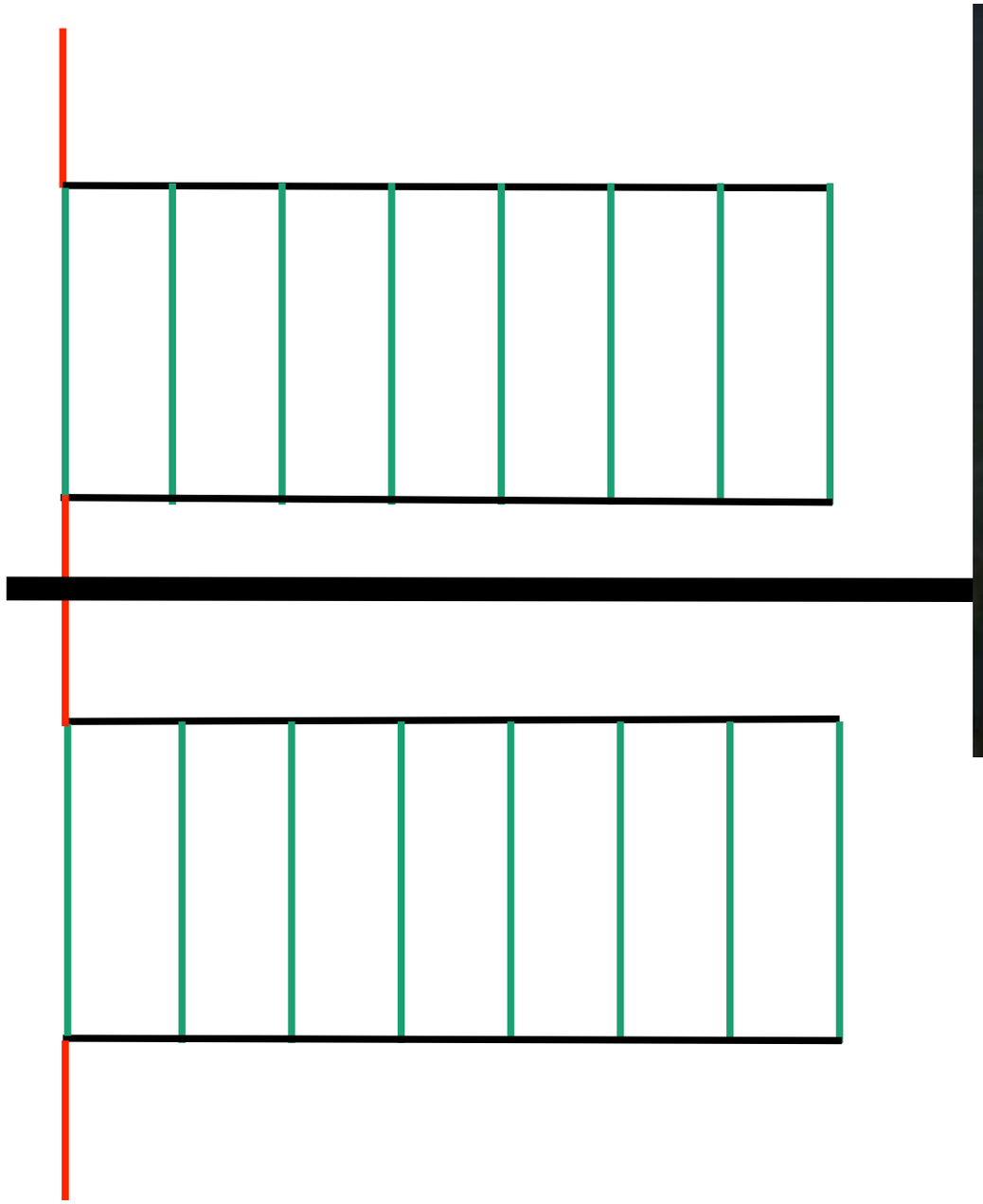


At the end of a parallel construct *every* thread waits until *every* thread executing the parallel construct has finished.

Barrier synchronization enforces this.

Raises issue of *load balance* and other inefficiencies.

By default, every parallel construct is followed by a *barrier*



At least not until the first loop finishes on *every* thread.

Barriers, good and bad

- Good because the synchronize data accesses across different loops
- Bad because every thread must wait for the slowest thread when executing a loop

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/ processors

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 that does bad things 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

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

```
#pragma omp parallel for private(j)
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]);
   }
}
```

Initialization of private variables

- The *firstprivate* gives the private variable the value the variable with the same name, and controlled by the master thread, has 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 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 private(tmp 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
- Our problem is to do this in parallel

```
t = 0;
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 (related to the earlier example?

Second way (and correct but slow)

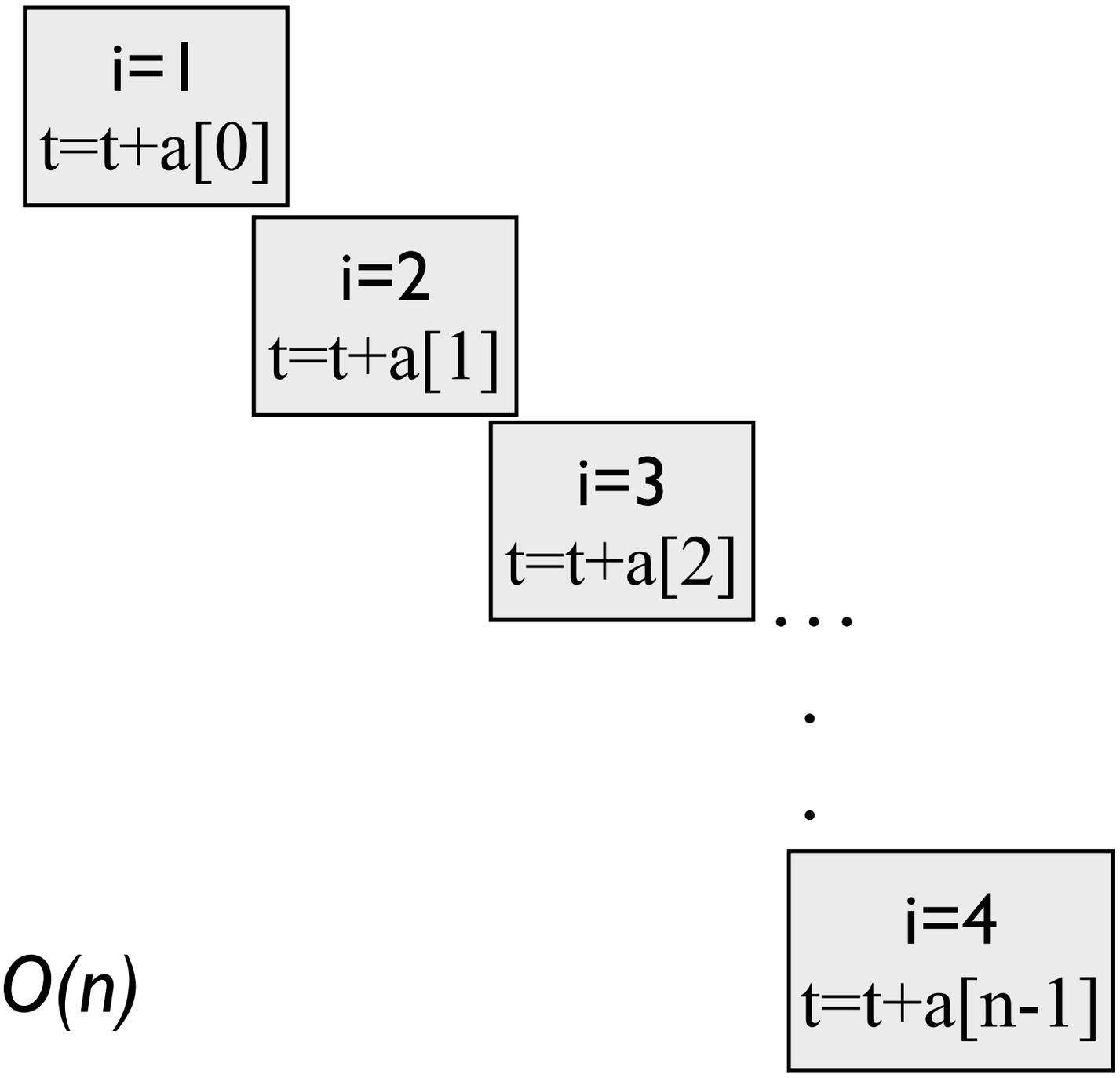
- use a *critical* section in the code
- executes the following (possible compound) statement atomically

```
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

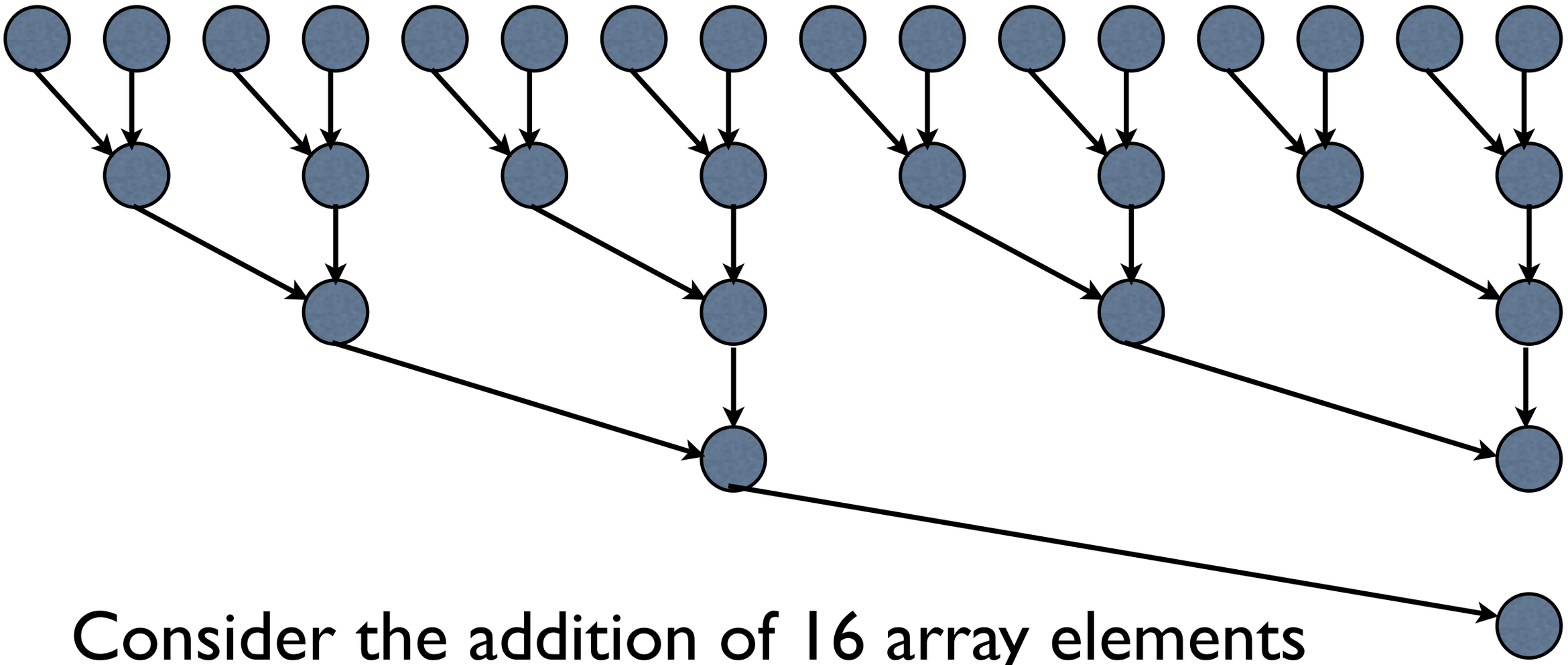
```
t = 0
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
   t = a[i];
}
t = t/n
```



time = O(n)

This is an example of a *reduction*

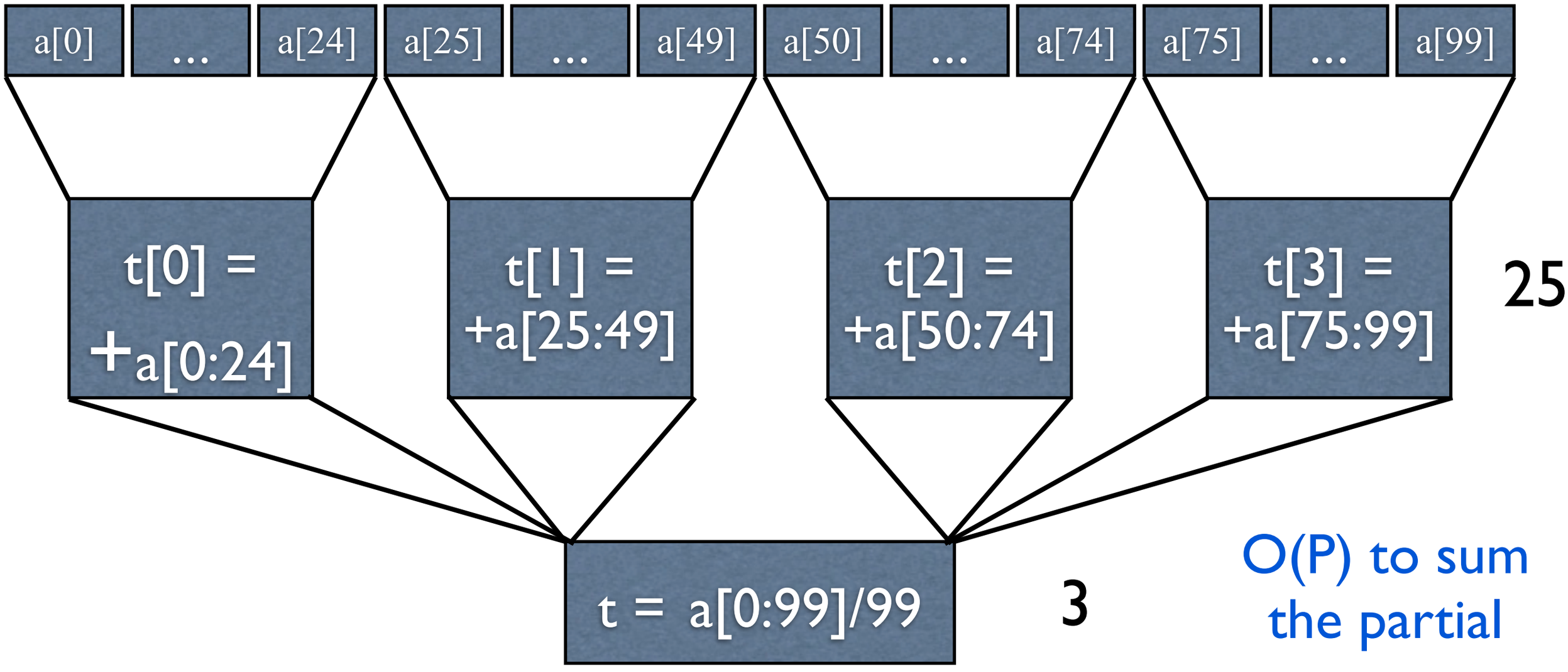
- Called a *reduction* because it takes something with *d* dimensions and reduces it to something with *d-k, k > 0* dimensions
- d and k is often 1, i.e. reductions often reduce a rank 1 array to a scalar.
- *Reductions on commutative operations can be done in parallel*



Consider the addition of 16 array elements to give a single sum. This takes $log_2 16 = 4$ time steps on 16 processors.

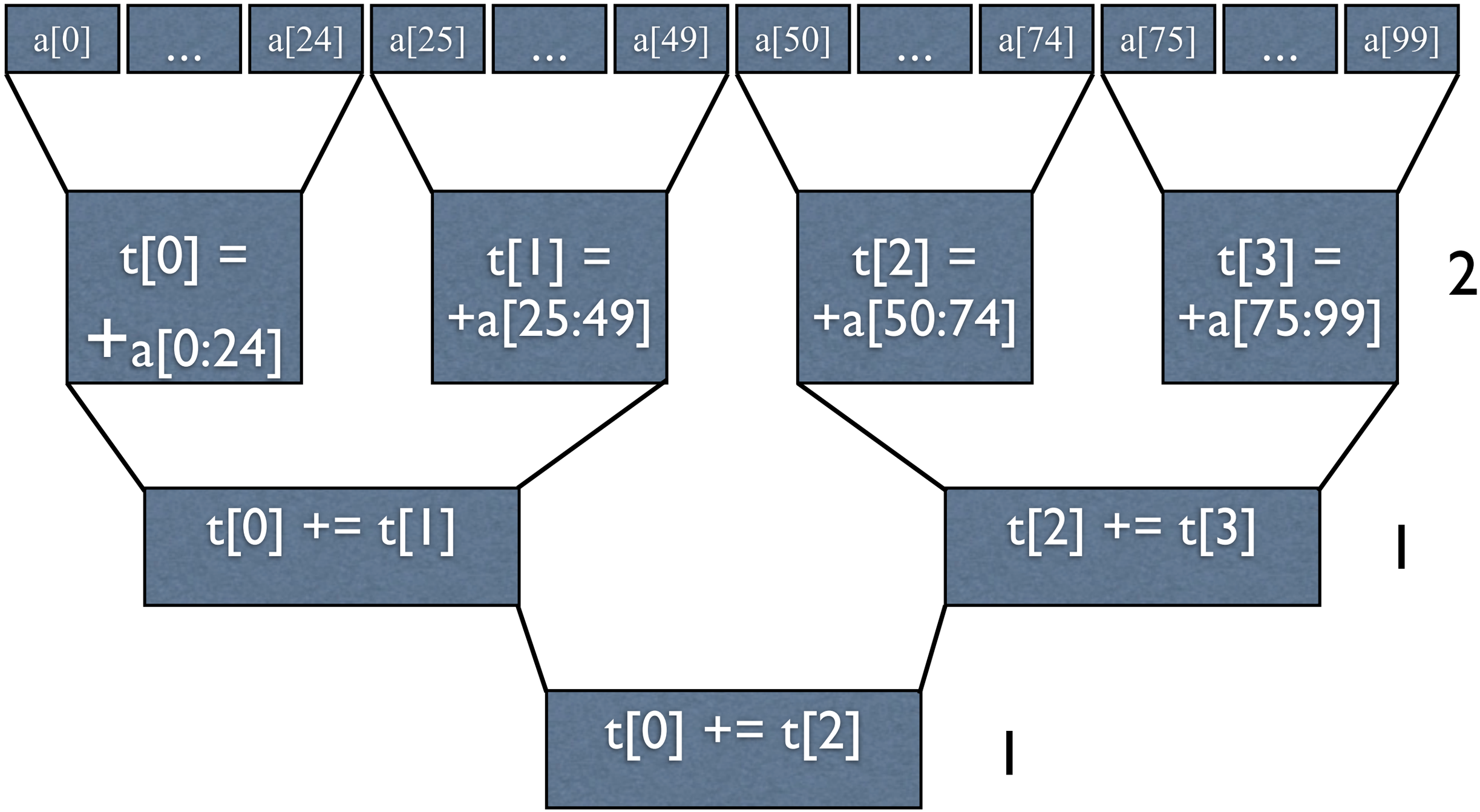
A parallel reduction

If more data than processors, each processor will compute a local sum before doing a reduction over these local sums on the *P* processors



O(P) to sum the partial sums

A better 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;
```

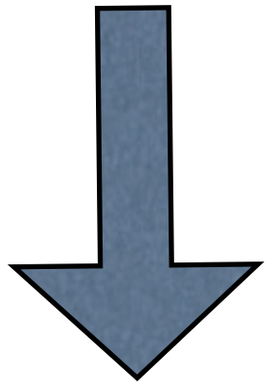
This is getting messy and we still are using a *O(#threads)* summation of the partial sums.

OpenMP provides a better way

- 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 and in good implementations does the more efficient sum

Dot product example

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



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

Operations supported by reduction

- +: sum
- *: multiplication
- &: bitwise AND
- |: bitwise OR
- ^: bitwise exclusive OR
- &&: logical AND
- ||: logical OR

A comment about recurrences

```
#pragma omp parallel for reduction(+:t)
// each element of b[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

$$s_t = (t\text{-}1)*ceil(n/\#threads)+i+1$$

- 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, very hard to do efficiently.
- Thus the OpenMP restriction on where t can appear

Improving performance

```
#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

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

Assign iterations to threads

- 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 statically.

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 *dynamic* chunks go to processors as work needed.

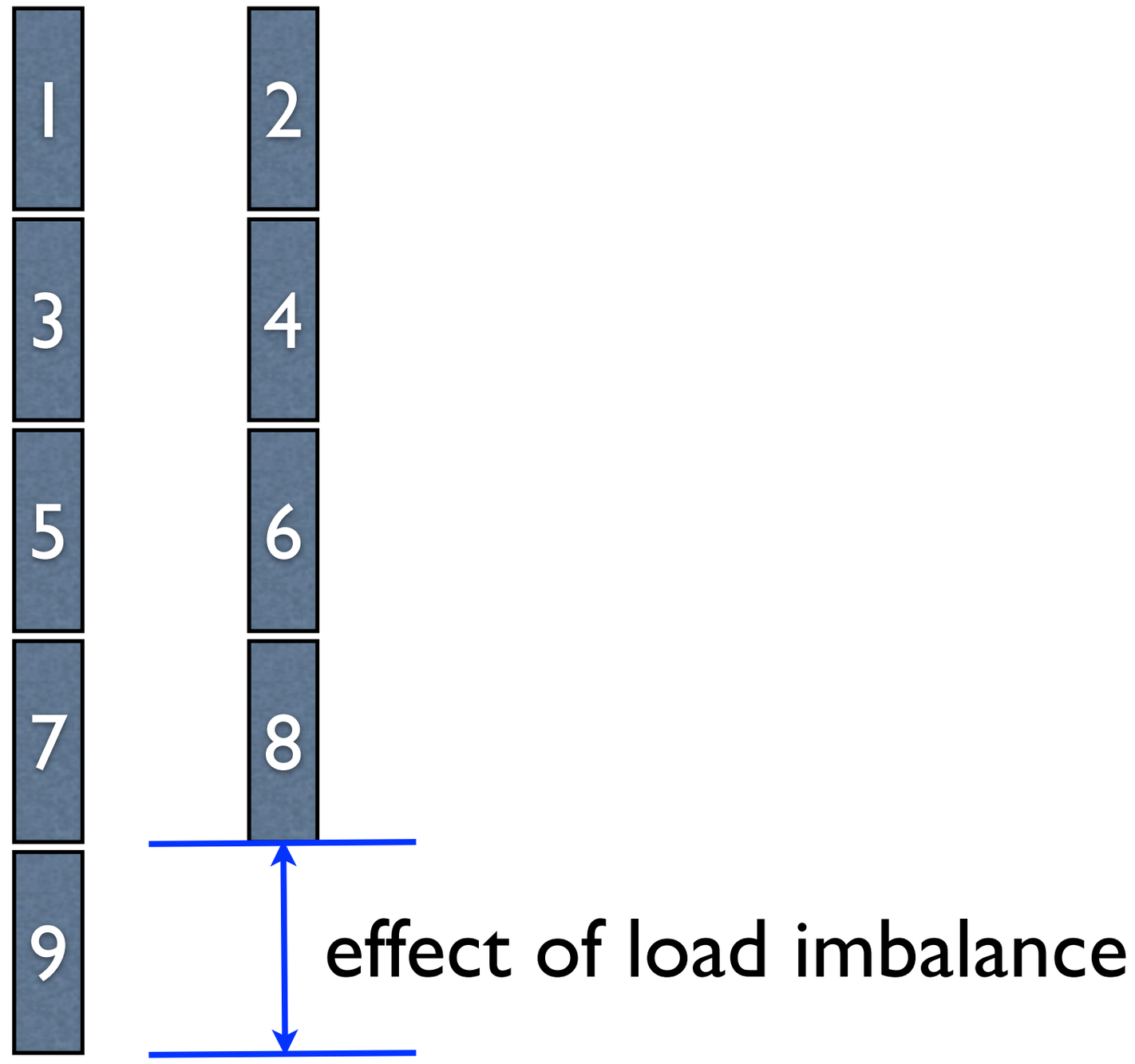
The schedule clause - dynamic

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

Dynamic with two threads example

Dynamic with two threads

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



The schedule clause - guided

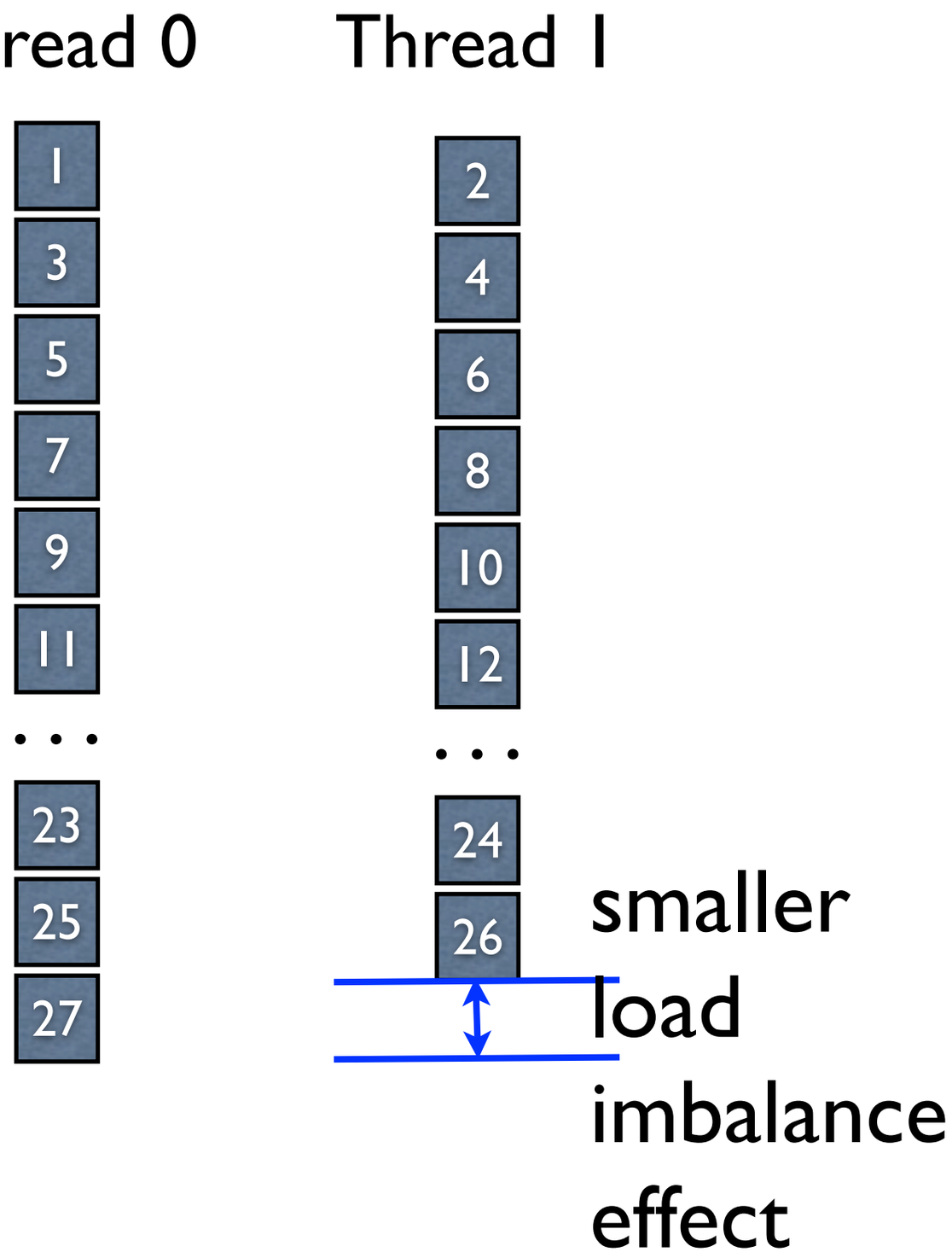
- schedule(*type[, chunk])* (type [,chunk]) is
  - (guided,chunk): uses *guided self scheduling* heuristic. Starts with big chunks and decreases to a minimum chunk size of *chunk*
  - runtime - type depends on value of OMP_SCHEDULE environment variable, e.g. setenv OMP_SCHEDULE=”static,1”

Guided with two threads

Large chunks have higher load imbalance, lower scheduling cost.

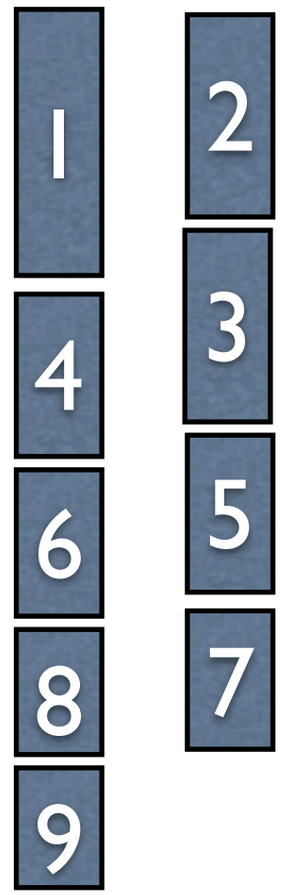
Small blocks have a smaller load imbalance, but with higher scheduling costs.

Would like the best of both methods. Guided gives this.



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.

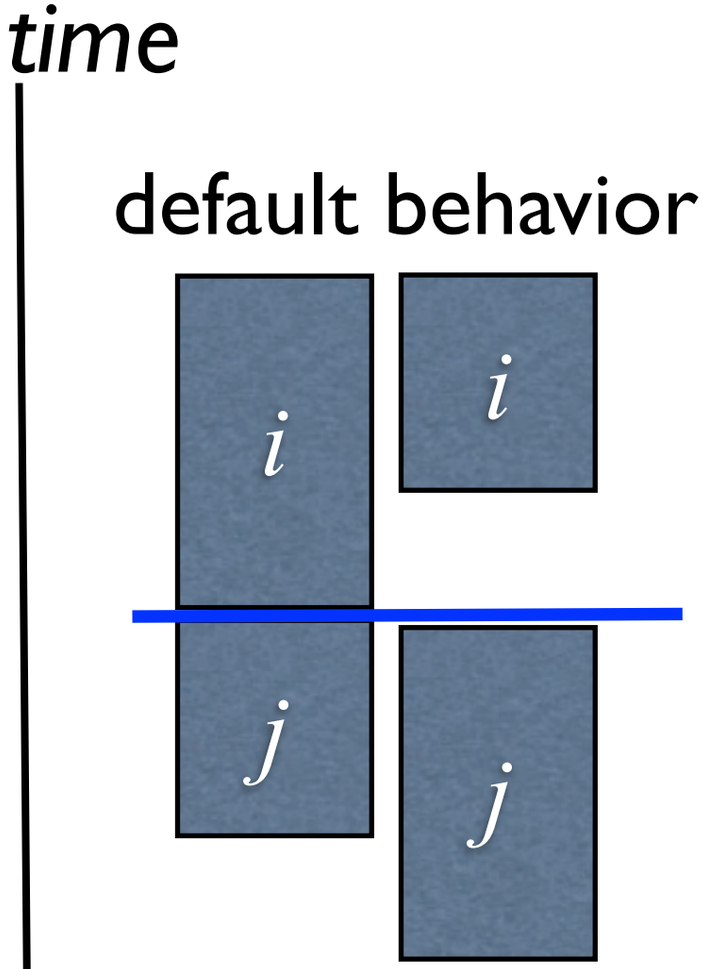


The nowait clause

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



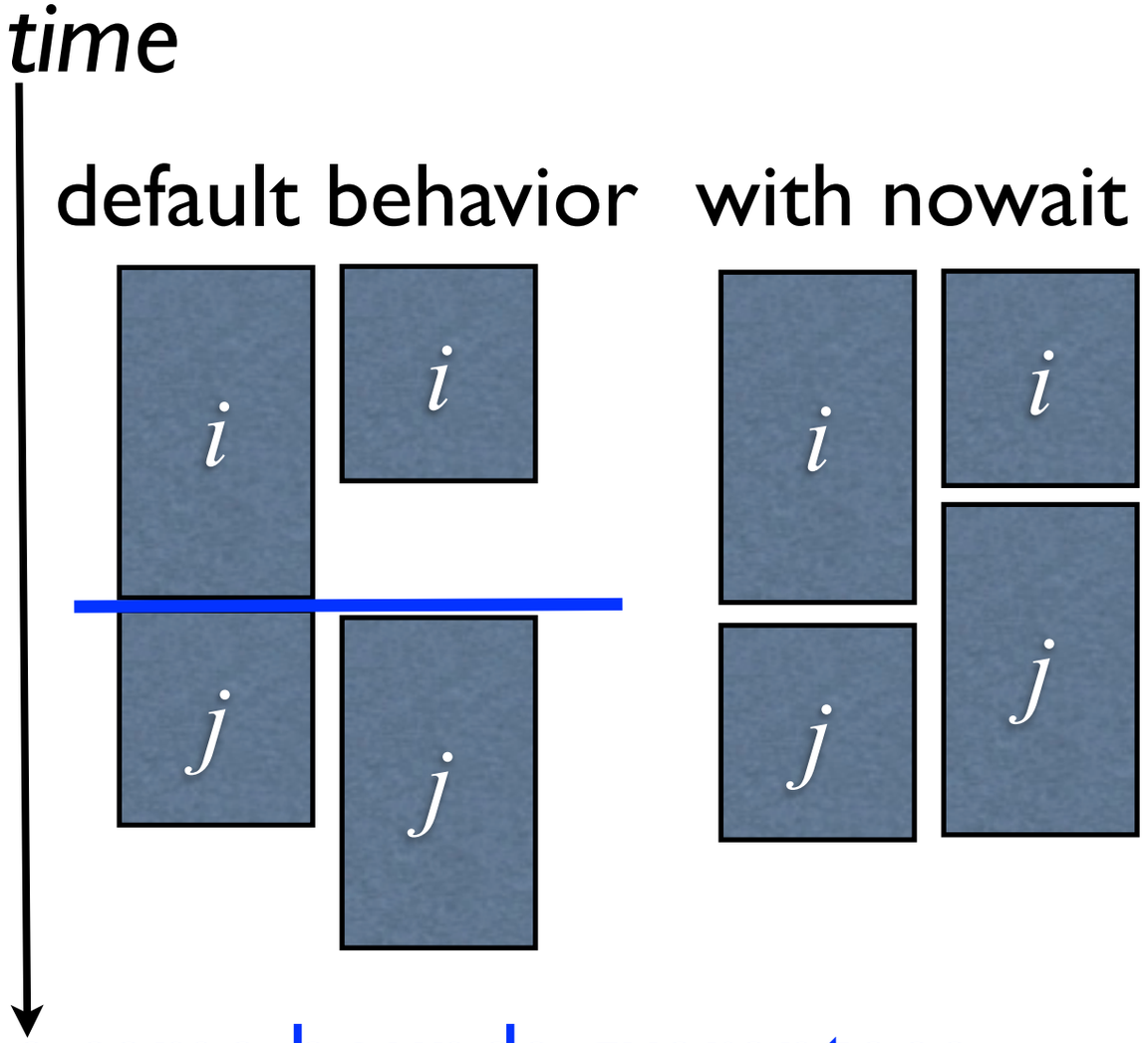
Only the static distribution with the same bounds guarantees the same thread will execute the same iterations from both loops.

The nowait clause

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

barrier here by default

```
#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.

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]
}
```

no barrier

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

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

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]
}
```

no barrier

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

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

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]
}
```

no barrier

```
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   d[i] = c[i] + a[i +1]*b[i]
}
```

If there are cross iteration dependences, the lack of a barrier can cause problems.

The single directive

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

master clause forces execution on the master thread.

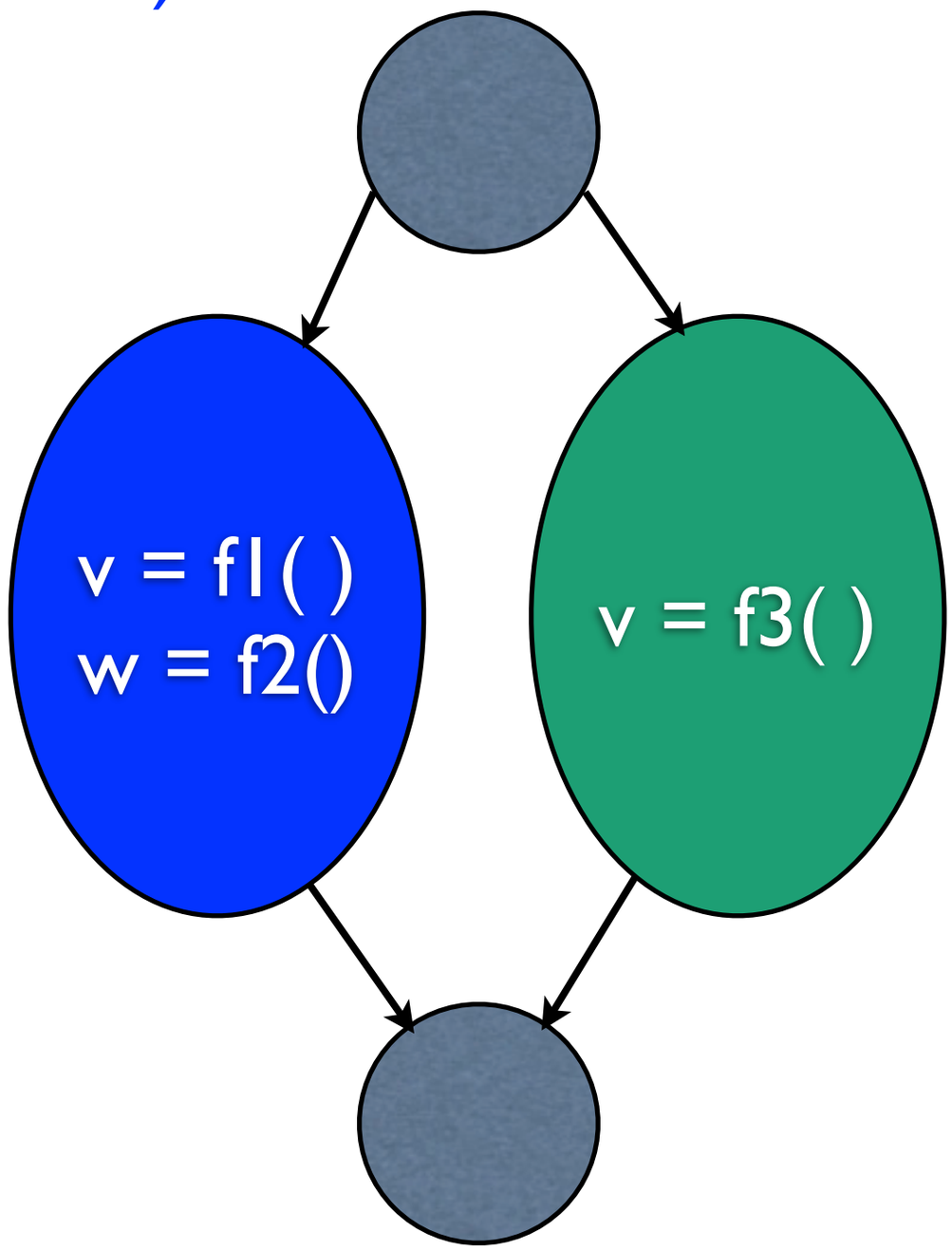
Requires statement following the pragma to be executed by a single available thread.

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

The sections pragma

Used to specify *task* (or functional) 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 out of q
- q needs to be thread safe

The parallel pragma

```
workItem w;
#pragma omp parallel private(w)
{
   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

OpenMP provides 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.

Program Translation for Microtasking Scheme (static scheduling)

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

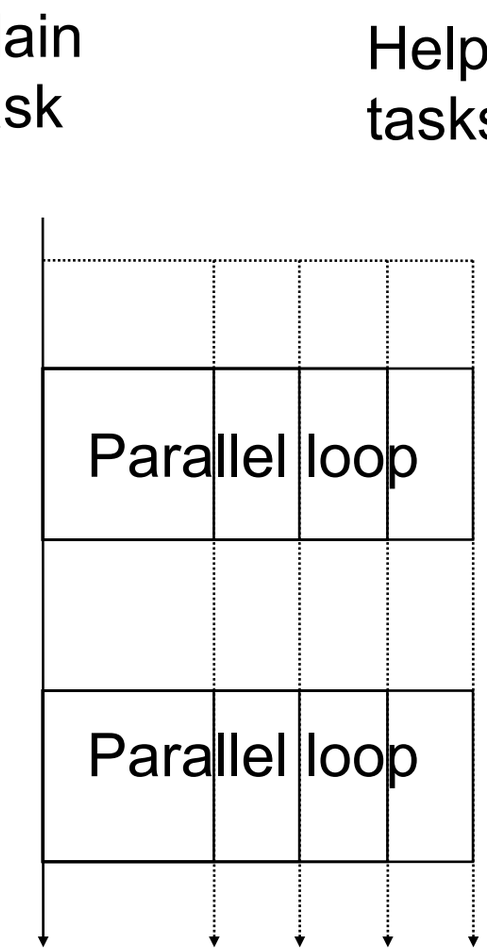


```
Subroutine x
…
call scheduler(1,n,a,b,loopsub)
…
END
```

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

Parallel ExecutionScheme

- Most widely used: Microtasking scheme



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] = + 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[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]
}
```

- 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

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];
}
```

Stupid but correct -- *critical* pragma allows only one thread to execute the next statement at a time. *Very inefficient -- but ok if you have enough work to make it worthwhile.*