Lecture 15: Worksharing, data directives in OpenMP

CMSE 822: Parallel Computing
Prof. Sean M. Couch

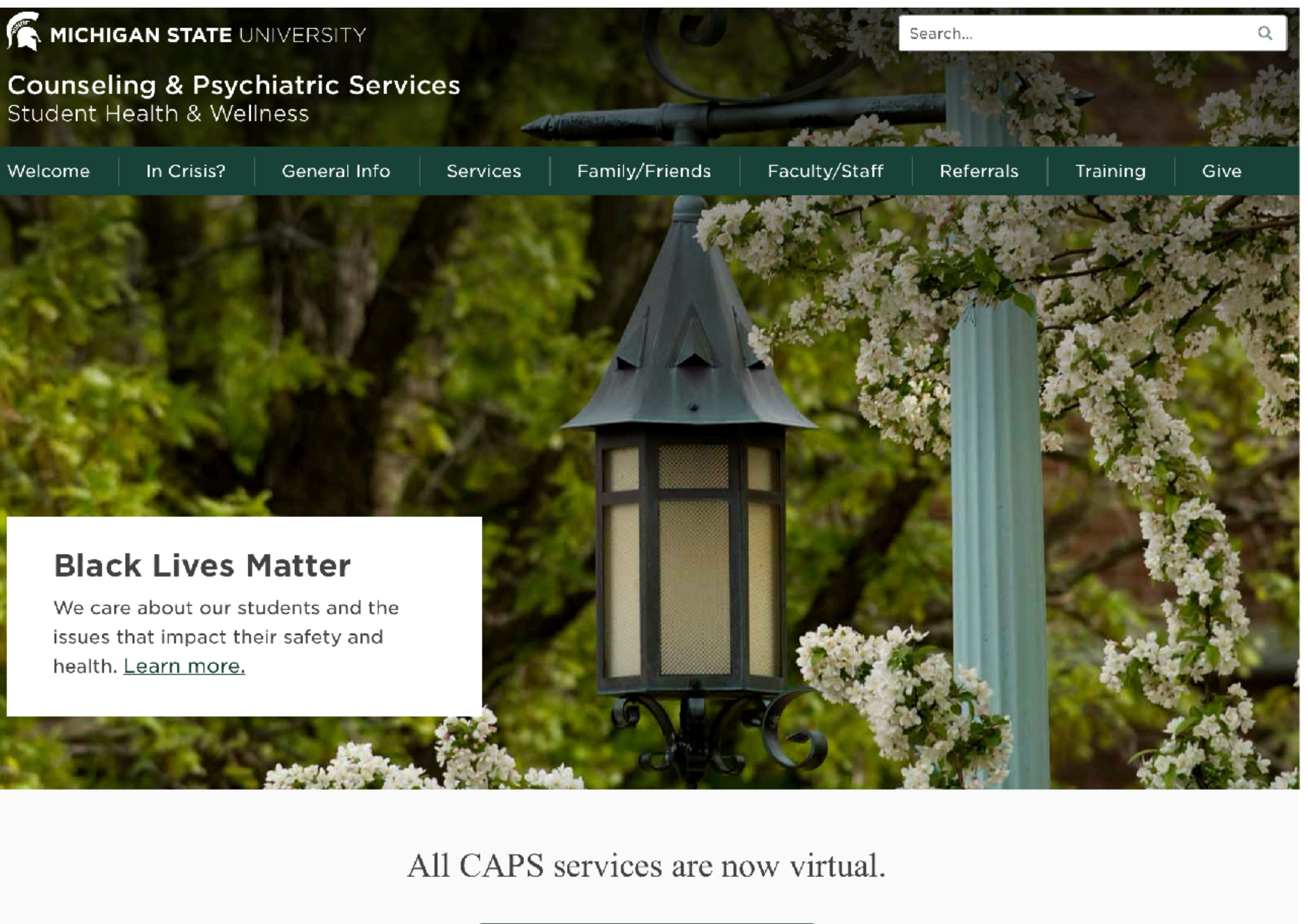
VOTE!

- michigan.gov/vote
- https://www.betterknowaballot.com
- Drop off ballot at clerk’s office!

© S.M. Couch

CAPS

caps.msu.edu



© S.M. Couch

Puppy time!

OpenMP Loops, Worksharing, Reductions

see https://computing.llnl.gov/tutorials/openMP

© S.M. Couch

Work-sharing Constructs

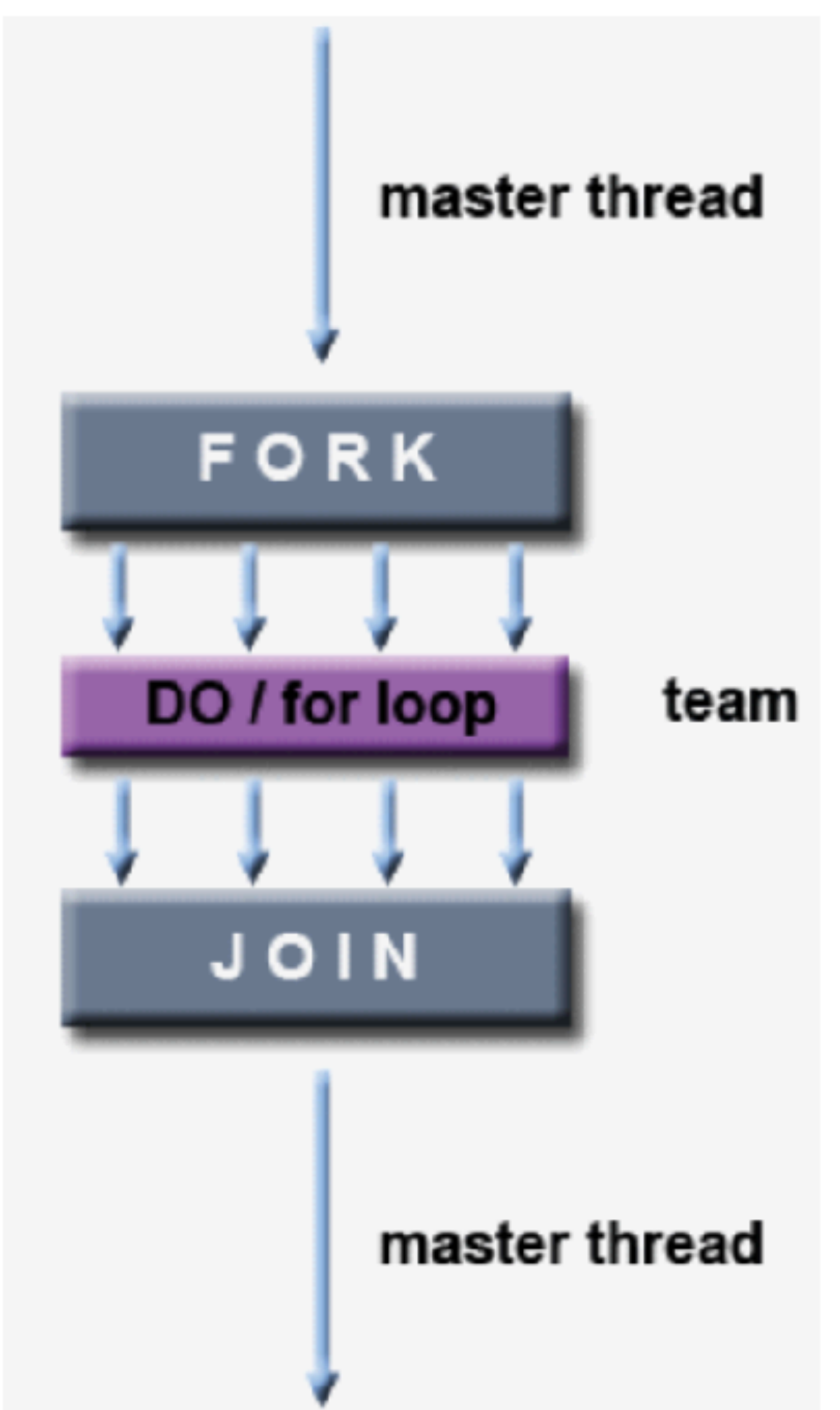
- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

© S.M. Couch

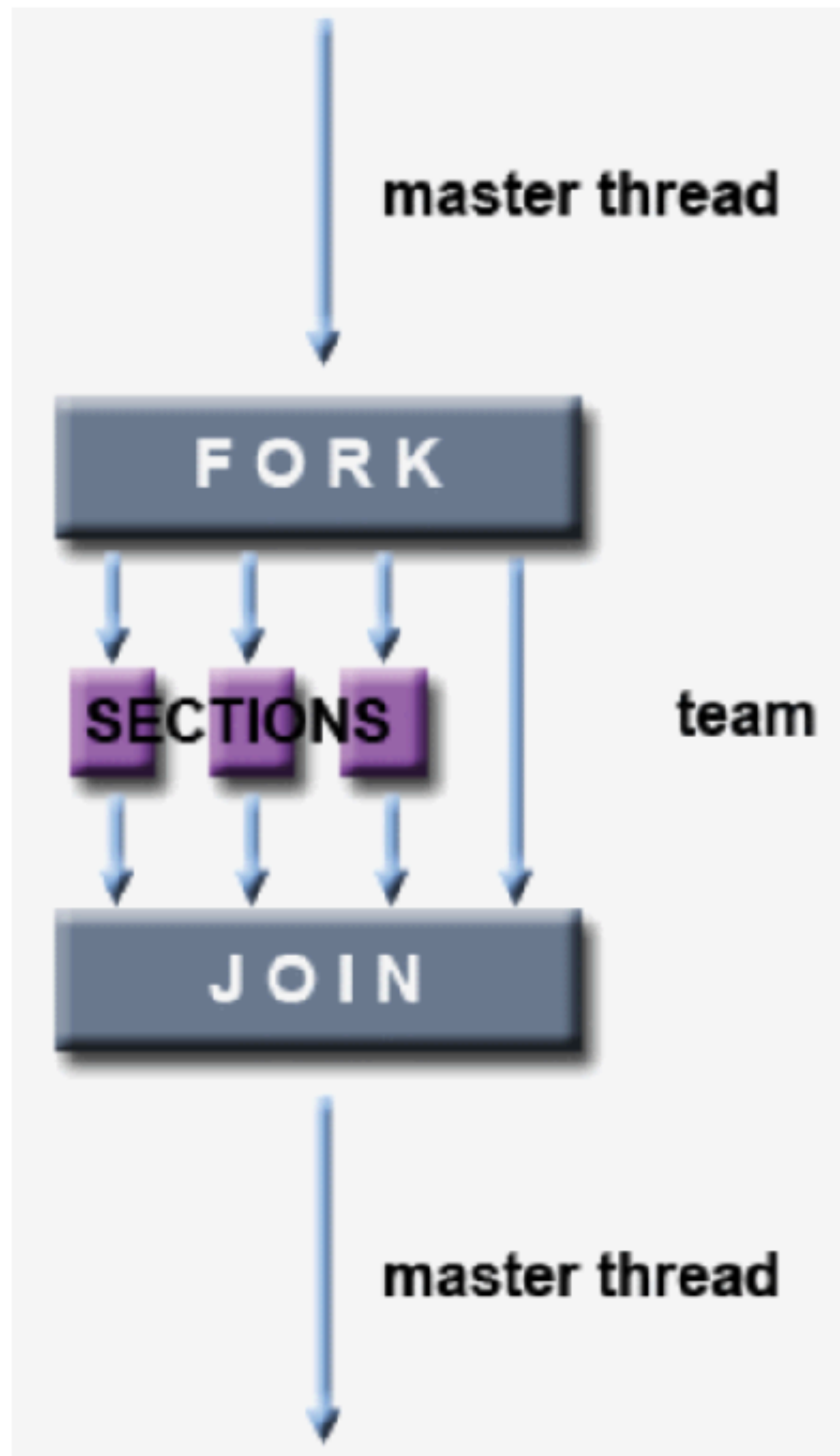
Types of Work-Sharing Constructs:

NOTE: The Fortran `workshare` construct is not shown here.

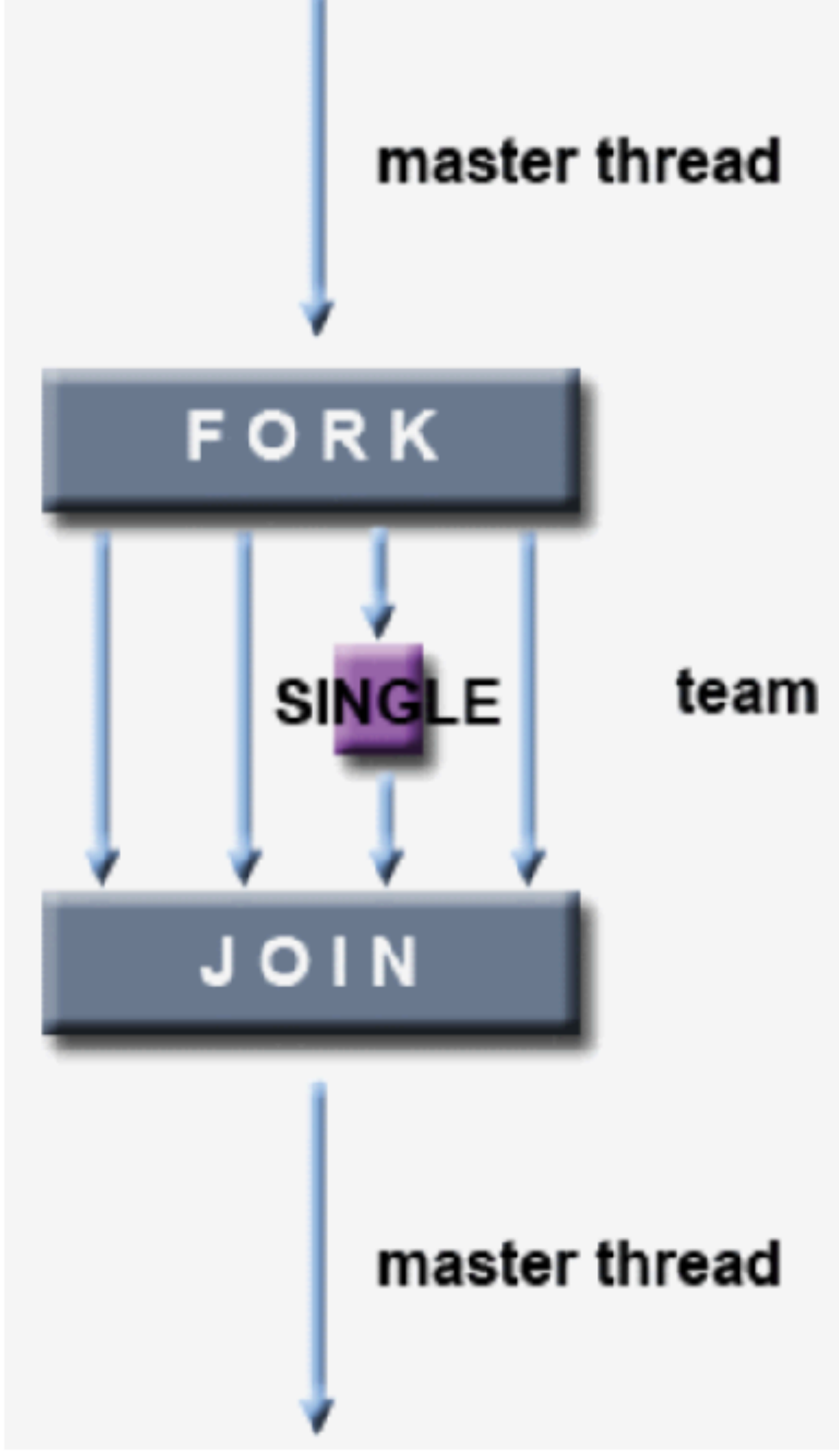
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



© S.M. Couch

Work-sharing Constructs

- **Restrictions:**
  - A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
  - Work-sharing constructs must be encountered by all members of a team or none at all
  - Successive work-sharing constructs must be encountered in the same order by all members of a team

© S.M. Couch

Work-sharing

DO / for Directive

Purpose:

- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

Format:

<table>
<tr><td>Fortran</td><td><pre>!$OMP DO [clause ...]
         SCHEDULE (type [,chunk])
         ORDERED
         PRIVATE (list)
         FIRSTPRIVATE (list)
         LASTPRIVATE (list)
         SHARED (list)
         REDUCTION (operator : list)
         COLLAPSE (n)

   do_loop

!$OMP END DO [NOWAIT]</pre></td></tr>
<tr><td>C/C++</td><td><pre>#pragma omp for [clause ...] newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop</pre></td></tr>
</table>

© S.M. Couch

Work-sharing

- **Clauses:**
    - **SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. For a discussion on how one type of scheduling may be more optimal than others, see http://openmp.org/forum/viewtopic.php?f=3&t=83.

© S.M. Couch

STATIC

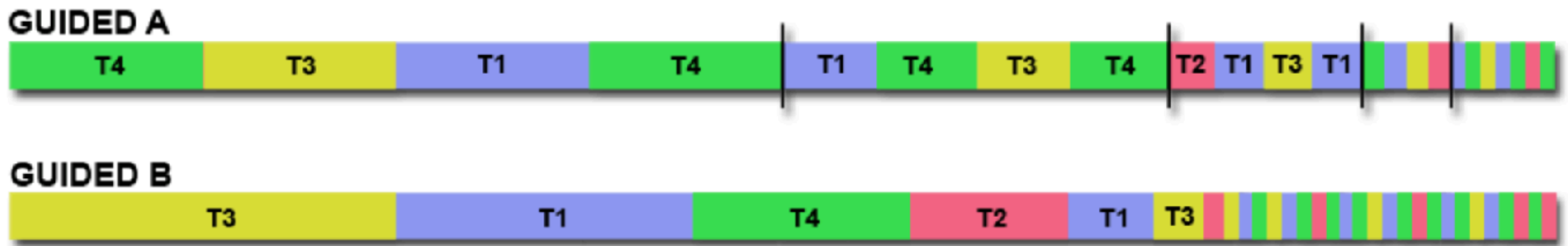
Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC

Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED

Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned.
Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.
The size of the initial block is proportional to: **`number_of_iterations / number_of_threads`**
Subsequent blocks are proportional to **`number_of_iterations_remaining / number_of_threads`**
The chunk parameter defines the minimum block size. The default chunk size is 1.
Note: compilers differ in how GUIDED is implemented as shown in the "Guided A" and "Guided B" examples below.



 © S.M. Couch

Work-sharing

- **Clauses:**
  - **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.
  - **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
  - **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the **`schedule`** clause. The order of the iterations in the collapsed iteration space is determined as though they were executed sequentially. May improve performance.
  - Other clauses are described in detail later, in the Data Scope Attribute Clauses section.
- **Restrictions:**
  - The DO loop can not be a DO WHILE loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
  - Program correctness must not depend upon which thread executes a particular iteration.
  - It is illegal to branch (goto) out of a loop associated with a DO/for directive.
  - The chunk size must be specified as a loop invariant integer expression, as there is no synchronization during its evaluation by different threads.
  - ORDERED, COLLAPSE and SCHEDULE clauses may appear once each.
  - See the OpenMP specification document for additional restrictions.

© S.M. Couch

Example: DO / for Directive

- Simple vector-add program
  - Arrays A, B, C, and variable N will be shared by all threads.
  - Variable I will be private to each thread; each thread will have its own unique copy.
  - The iterations of the loop will be distributed dynamically in CHUNK sized pieces.
  - Threads will not synchronize upon completing their individual pieces of work (NOWAIT).

```
1  #include <omp.h>
2  #define N 1000
3  #define CHUNKSIZE 100
4
5  main(int argc, char *argv[]) {
6
7  int i, chunk;
8  float a[N], b[N], c[N];
9
10 /* Some initializations */
11 for (i=0; i < N; i++)
12   a[i] = b[i] = i * 1.0;
13 chunk = CHUNKSIZE;
14
15 #pragma omp parallel shared(a,b,c,chunk) private(i)
16   {
17
18   #pragma omp for schedule(dynamic,chunk) nowait
19   for (i=0; i < N; i++)
20     c[i] = a[i] + b[i];
21
22   }   /* end of parallel region */
23
24 }
```


© S.M. Couch

Sections

- **Purpose:**
  - The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
  - Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.
- **Clauses:**
  - There is an implied barrier at the end of a SECTIONS directive, unless the `NOWAIT/nowait` clause is used.
- **Restrictions:**
  - It is illegal to branch (goto) into or out of section blocks.
  - SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive (no orphan SECTIONs).

Fortran

```
!$OMP SECTIONS [clause ...]
               PRIVATE (list)
               FIRSTPRIVATE (list)
               LASTPRIVATE (list)
               REDUCTION (operator | intrinsic : list)

!$OMP  SECTION

   block

!$OMP  SECTION

    block

!$OMP END SECTIONS  [ NOWAIT ]
```

C/C++

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   newline

     structured_block

  #pragma omp section   newline

     structured_block

  }
```


© S.M. Couch

Sections

```c
#include <omp.h>
#define N 1000

main(int argc, char *argv[]) {

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel region */

}
```

 © S.M. Couch

Single

- **Purpose:**
  - The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
  - May be useful when dealing with sections of code that are not thread safe (such as I/O)
- **Format:**

Fortran	`!$OMP SINGLE [clause ...]`
	`PRIVATE (list)`
	`FIRSTPRIVATE (list)`
	
	`block`
	
	`!$OMP END SINGLE [ NOWAIT ]`

C/C++	`#pragma omp single [clause ...]  newline`
	`private (list)`
	`firstprivate (list)`
	`nowait`
	
	`structured_block`

© S.M. Couch

Data Scope Attribute Clauses

- Also called **Data-sharing** Attribute Clauses
- An important consideration for OpenMP programming is the understanding and use of data scoping
- Because OpenMP is based upon the shared memory programming model, most variables are shared by default
- Global variables include:
    - Fortran: COMMON blocks, SAVE variables, MODULE variables
    - C: File scope variables, static
- Private variables include:
    - Loop index variables
    - Stack variables in subroutines called from parallel regions
    - Fortran: Automatic variables within a statement block
- The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:
    - PRIVATE
    - FIRSTPRIVATE
    - LASTPRIVATE
    - SHARED
    - DEFAULT
    - REDUCTION
    - COPYIN

© S.M. Couch

Data Scope Attribute Clauses

- Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.
- These constructs provide the ability to control the data environment during execution of parallel constructs.
    - They define how and which data variables in the serial section of the program are transferred to the parallel regions of the program (and back)
    - They define which variables will be visible to all threads in the parallel regions and which variables will be privately allocated to all threads.
- Data Scope Attribute Clauses are effective only within their lexical/static extent.
- **Important:** Please consult the latest OpenMP specs for important details and discussion on this topic.

© S.M. Couch

Data Scope Attribute Clauses

PRIVATE Clause

Purpose:

- The PRIVATE clause declares variables in its list to be private to each thread.

Format:

Fortran	**PRIVATE *(list)***
C/C++	**private *(list)***

Notes:

- PRIVATE variables behave as follows:
    - A new object of the same type is declared once for each thread in the team
    - All references to the original object are replaced with references to the new object
    - Should be assumed to be uninitialized for each thread

© S.M. Couch

Data Scope Attribute Clauses

SHARED Clause

Purpose:

- The SHARED clause declares variables in its list to be shared among all threads in the team.

Format:

Fortran	`SHARED (list)`
C/C++	`shared (list)`

Notes:

- A shared variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

© S.M. Couch

Data Scope Attribute Clauses

DEFAULT Clause

Purpose:

- The DEFAULT clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.

Format:

Fortran	`DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)`
C/C++	`default (shared | none)`

Notes:

- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses
- The C/C++ OpenMP specification does not include private or firstprivate as a possible default. However, actual implementations may provide this option.
- Using NONE as a default requires that the programmer explicitly scope all variables.

Restrictions:

- Only one DEFAULT clause can be specified on a PARALLEL directive

© S.M. Couch

Data Scope Attribute Clauses

FIRSTPRIVATE Clause

Purpose:

- The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

Format:

Fortran	`FIRSTPRIVATE (list)`
C/C++	`firstprivate (list)`

Notes:

- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

© S.M. Couch

Data Scope Attribute Clauses

LASTPRIVATE Clause

Purpose:

- The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.

Format:

Fortran	`LASTPRIVATE (list)`
C/C++	`lastprivate (list)`

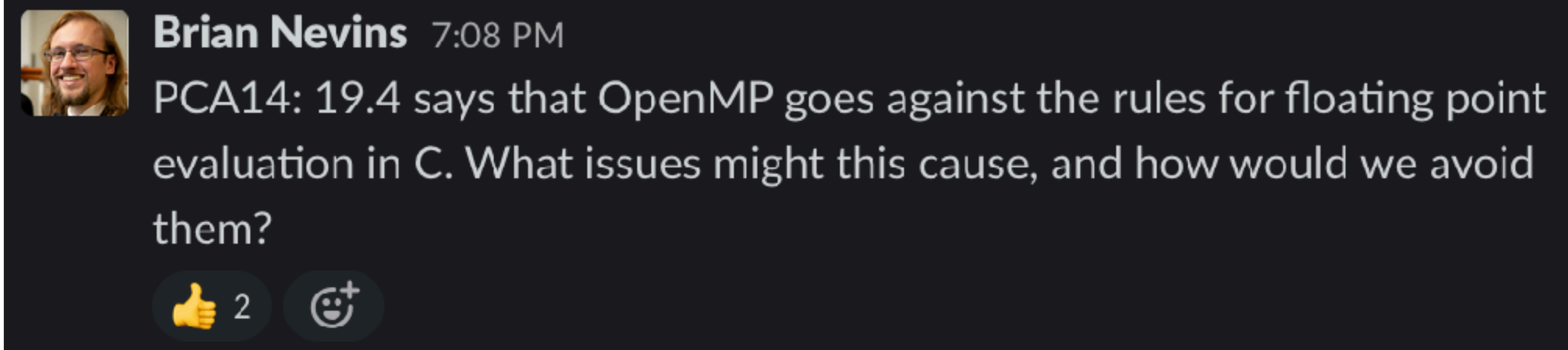
Notes:

- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.

  For example, the team member which executes the final iteration for a DO section, or the team member which does the last SECTION of a SECTIONS context performs the copy with its own values

© S.M. Couch

PCA Questions



- Addition is non-associative, numerically…
- In OpenMP, order of additions is not guaranteed

© S.M. Couch

PCA Questions

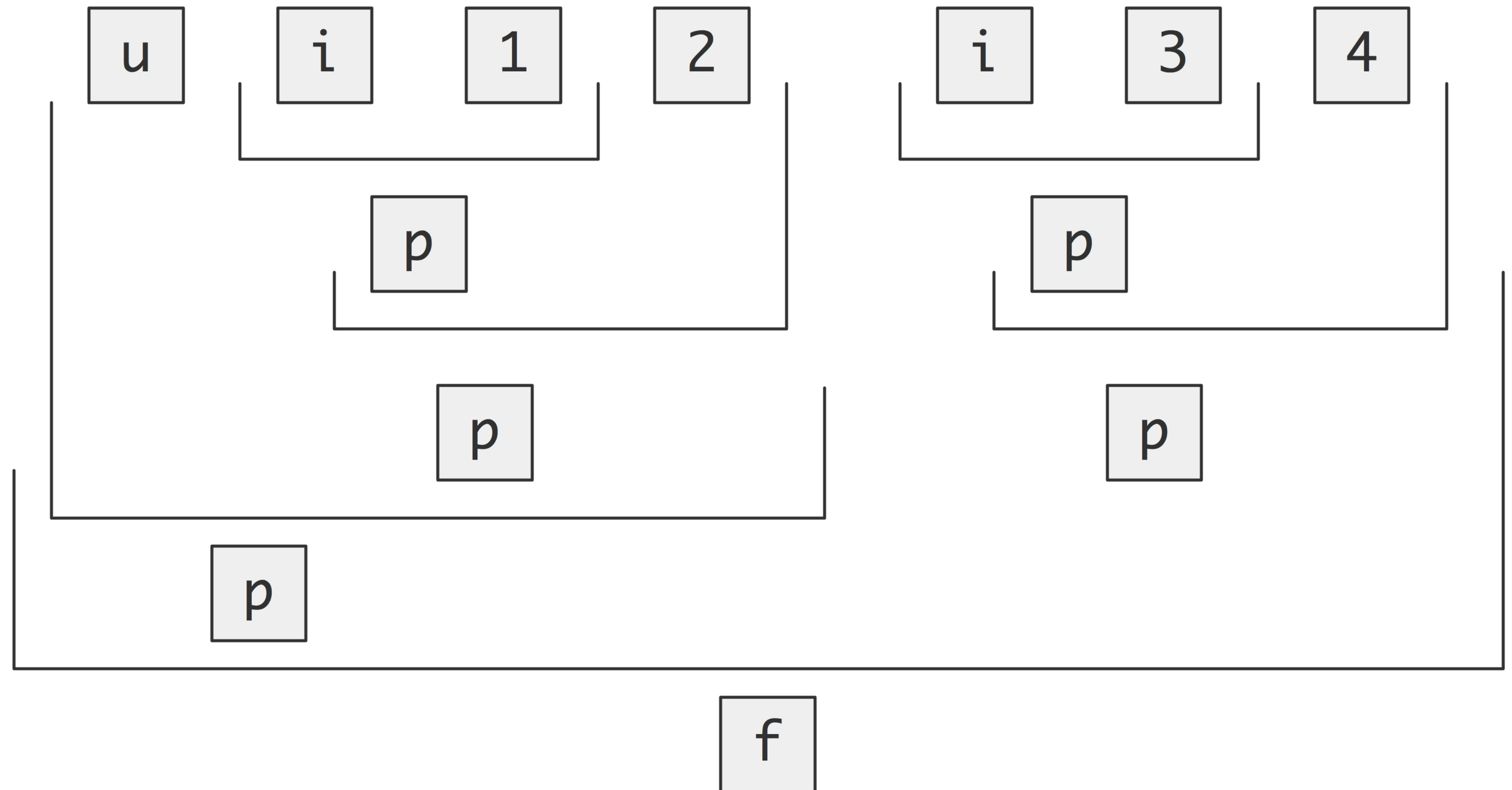
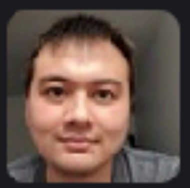


Figure 19.1: Reduction of four items on two threads, taking into account initial values.

Figure 19.1 illustrates this, where `1,2,3,4` are four data items, `i` is the OpenMP initialization, and `u` is the user initialization; each `p` stands for a partial reduction value. The figure is based on execution using two threads.



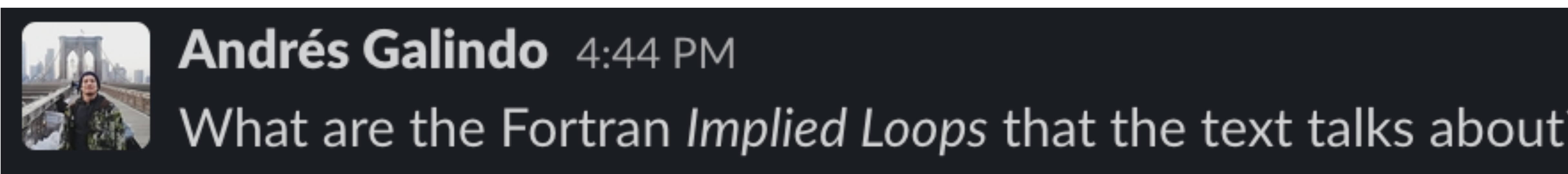
Matthew Zeilbeck 3:44 PM

PCA14: How does a reduction actually get computed in OpenMP? Even though you declare a reduction variable outside of the parallel code and before `#pragma omp ... reduction(...)`, each thread gets a private copy of a variable of the same name that is initialized to some value depending on the operation. Then, according to figure 19.1, it looks like the reduction operation is computed pairwise on each variable, starting with each thread's variable (e.g. `a[i]`) and the private reduction variable. Then the function is computed on (`a[0]`, `a[1]`), (`a[2]`, `a[3]`), and so on, cutting the number of remaining variables in half each time. Is this how OpenMP does reductions? Is this what MPI is doing behind the scenes? Lastly, does this take an O(n) runtime into an O(log n) runtime?

```
  x = init_x
#pragma omp parallel for reduction(min:x)
  for (int i=0; i<N; i++)
    x = min(x,data[i]);
```

- Can be compiler-specific!

PCA Questions

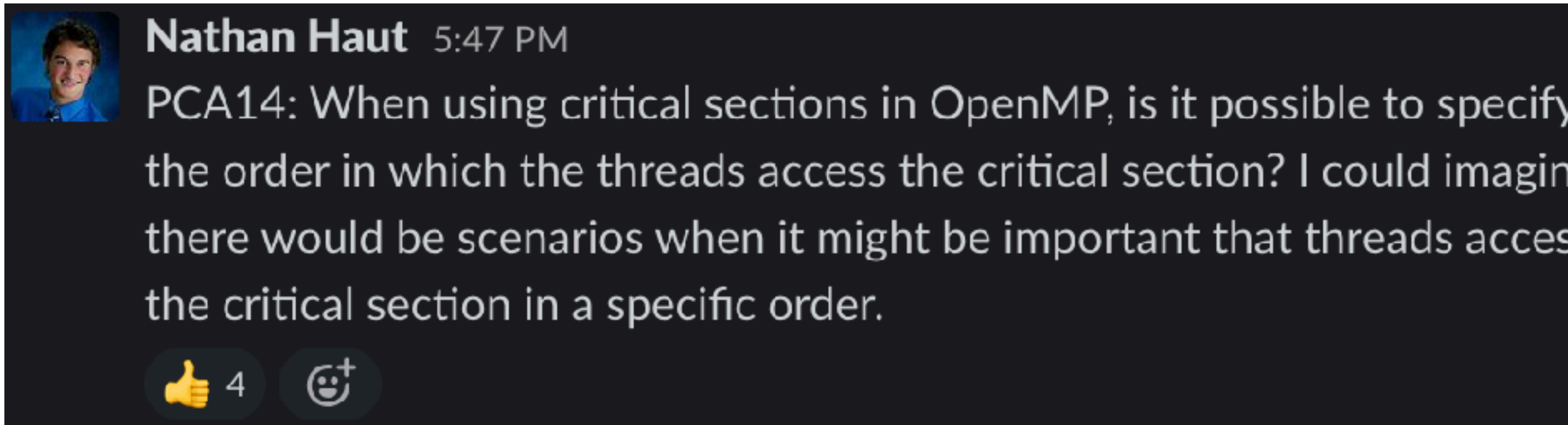


- Array operations:

```
invSumAlpha = 1./(Alpha5(:,1)+Alpha5(:,2)+Alpha5(:,3))
omega(:,1)  = Alpha5(:,1)*invSumAlpha
omega(:,2)  = Alpha5(:,2)*invSumAlpha
omega(:,3)  = Alpha5(:,3)*invSumAlpha
```

 © S.M. Couch

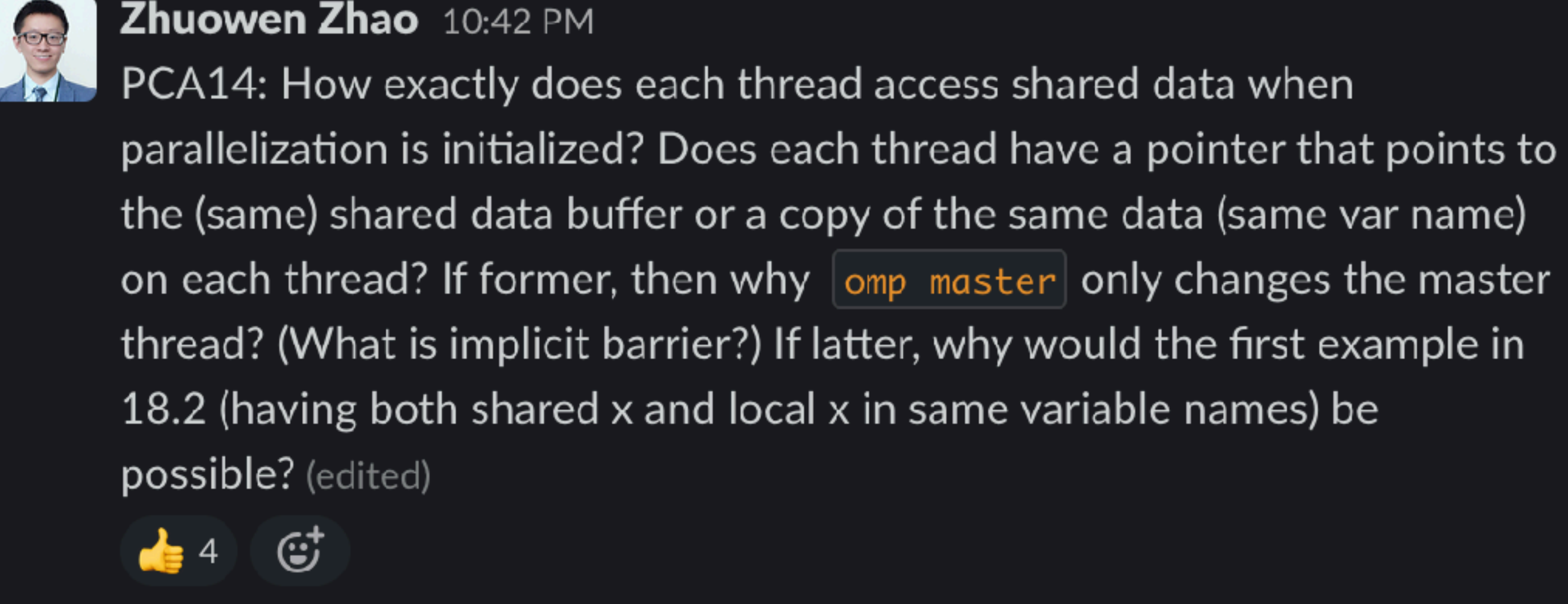
PCA Questions



- No…
- See ORDERED directive for loops, however.

© S.M. Couch

PCA Questions




```
  int x = 5;
#pragma omp parallel
  {
    int x; x = 3;
    printf("local: x is %d\n",x);
  }
```

After the parallel region the outer variable `x` will still have the value `5`: there is no *storage association* between the private variable and global one.

The `private` directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

```
  int x = 5;
#pragma omp parallel private(x)
  {
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
  }
  printf("after: x is %d\n",x); // also dangerous
```

 © S.M. Couch

threadprivate

Purpose:

- The THREADPRIVATE directive specifies that variables are replicated, with each thread having its own copy.
- Can be used to make global file scope variables (C/C++/Fortran) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.

Format:

Fortran	`!$OMP THREADPRIVATE (list)`
C/C++	`#pragma omp threadprivate (list)`

Notes:

- The directive must appear after the declaration of listed variables/common blocks. Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.

- On first entry to a parallel region, data in THREADPRIVATE variables and common blocks should be assumed undefined, unless a COPYIN clause is specified in the PARALLEL directive

- THREADPRIVATE variables differ from PRIVATE variables (discussed later) because they are able to persist between different parallel regions of a code.

© S.M. Couch

threadprivate

```
Output:

1st Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 2:   a,b,x= 2 2 3.200000
Thread 3:   a,b,x= 3 3 4.300000
Thread 1:   a,b,x= 1 1 2.100000
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 3:   a,b,x= 3 0 4.300000
Thread 1:   a,b,x= 1 0 2.100000
Thread 2:   a,b,x= 2 0 3.200000
```

```
1  #include <omp.h>
2
3  int  a, b, i, tid;
4  float x;
5
6  #pragma omp threadprivate(a, x)
7
8  main(int argc, char *argv[]) {
9
10   /* Explicitly turn off dynamic threads */
11   omp_set_dynamic(0);
12
13   printf("1st Parallel Region:\n");
14 #pragma omp parallel private(b,tid)
15   {
16   tid = omp_get_thread_num();
17   a = tid;
18   b = tid;
19   x = 1.1 * tid +1.0;
20   printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
21   }  /* end of parallel region */
22
23   printf("************************************\n");
24   printf("Master thread doing serial work here\n");
25   printf("************************************\n");
26
27   printf("2nd Parallel Region:\n");
28 #pragma omp parallel private(tid)
29   {
30   tid = omp_get_thread_num();
31   printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
32   }  /* end of parallel region */
33
34 }
```


© S.M. Couch