OpenMP

The main point

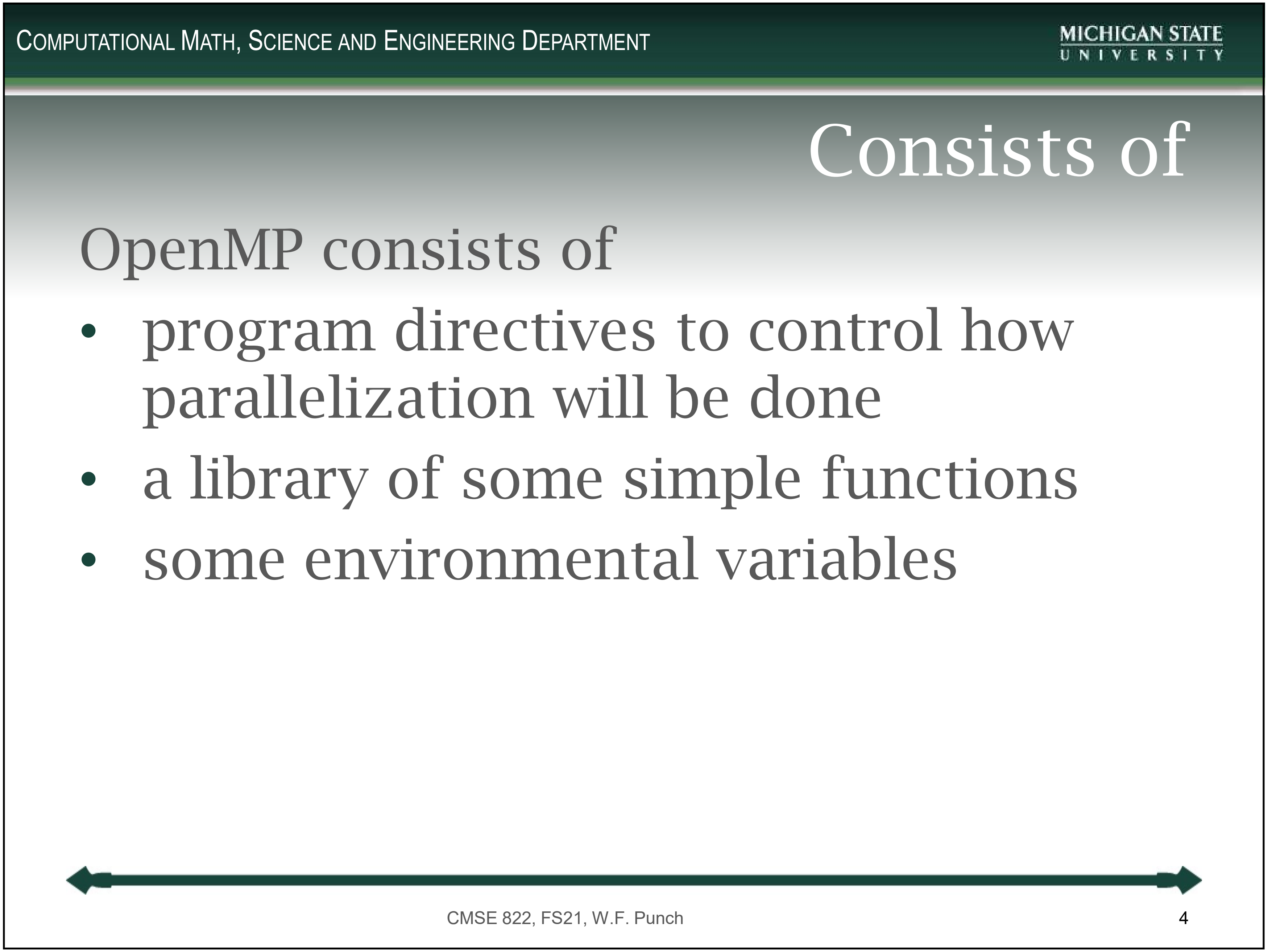
OpenMP (Open Multiprocessing) is an Application Programming Interface (API) that makes it easier to write threaded programs, programs that use shared memory.

Interfaces directly with C/C++ and Fortran

references

Take a look at www.openmp.org

- OpenMP 5.1 is the most recent version (2020)
- Previous to that 4.5, (2015)
  - for Ubuntu 20.04 LTS looks like we get that one.



Consists of

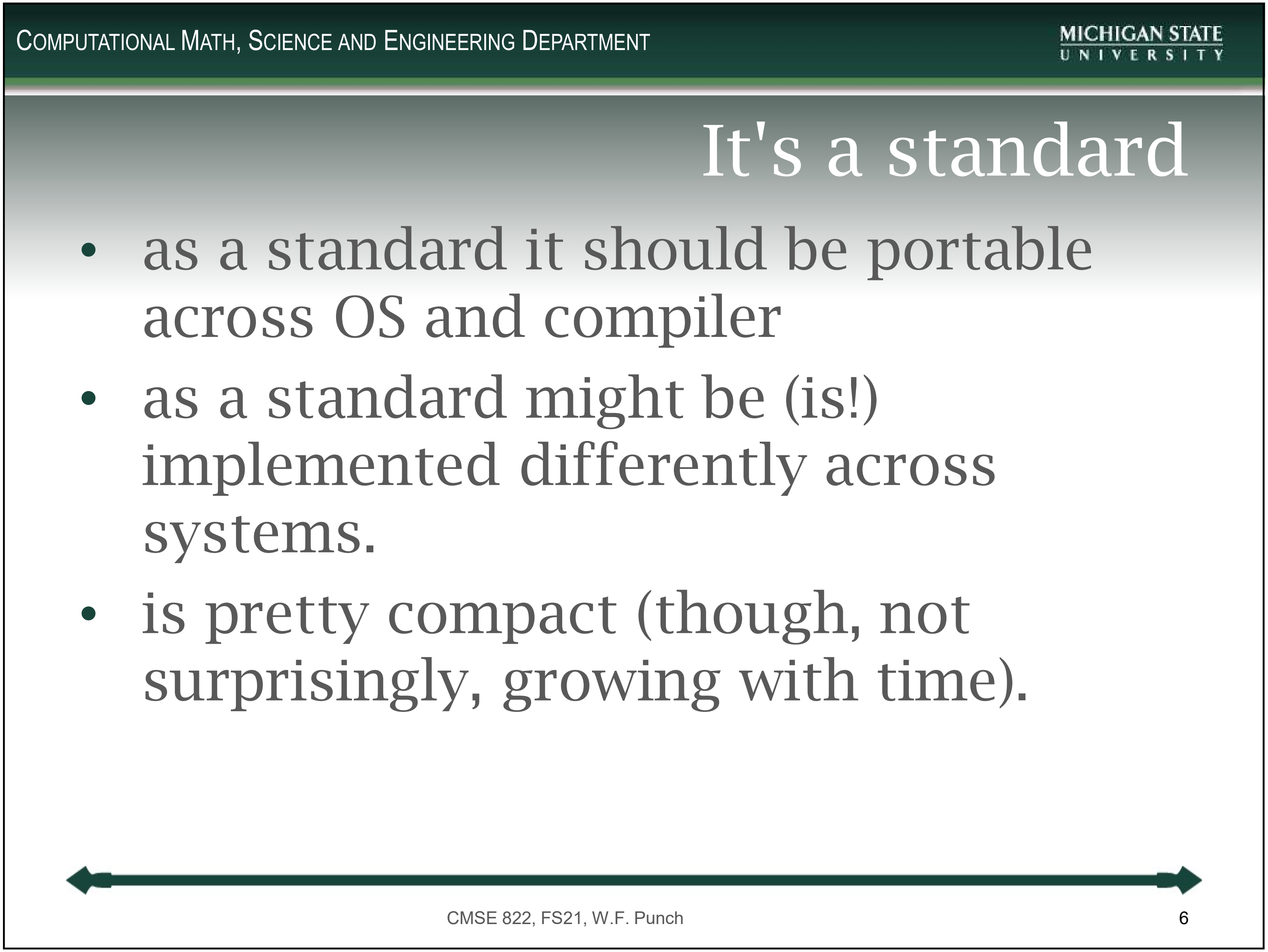
OpenMP consists of

- program directives to control how parallelization will be done
- a library of some simple functions
- some environmental variables

OpenMP, not OpenMPI

You can easily get confused here, but:

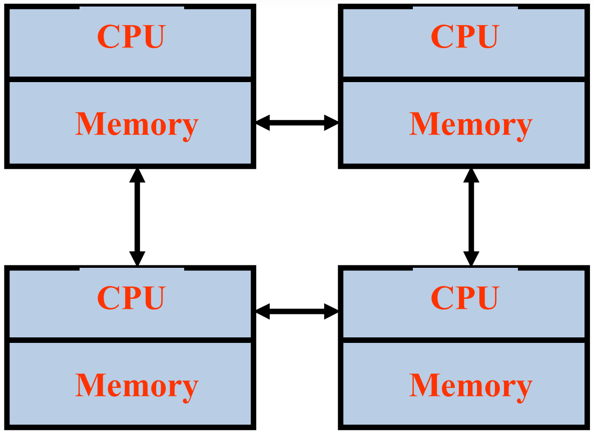
- OpenMP is a programming standard for shared memory/thread programming
- OpenMPI is a ***particular*** implementation of the MPI standard



It's a standard

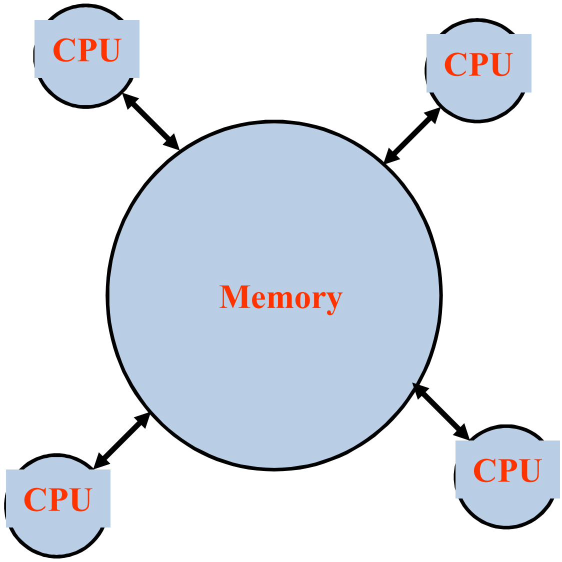
- as a standard it should be portable across OS and compiler
- as a standard might be (is!) implemented differently across systems.
- is pretty compact (though, not surprisingly, growing with time).

Shared Memory type



Distributed

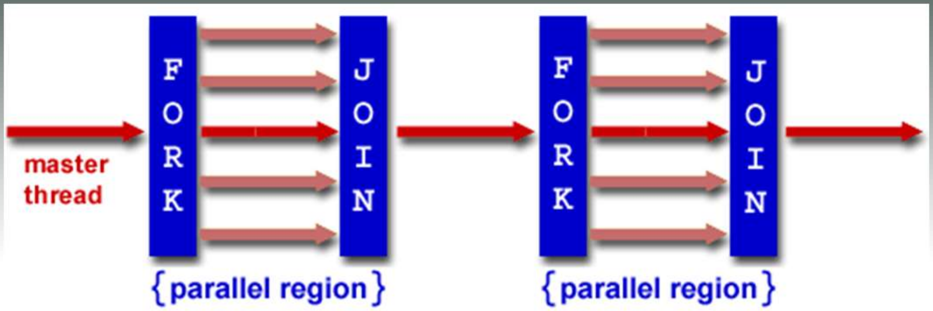
Shared



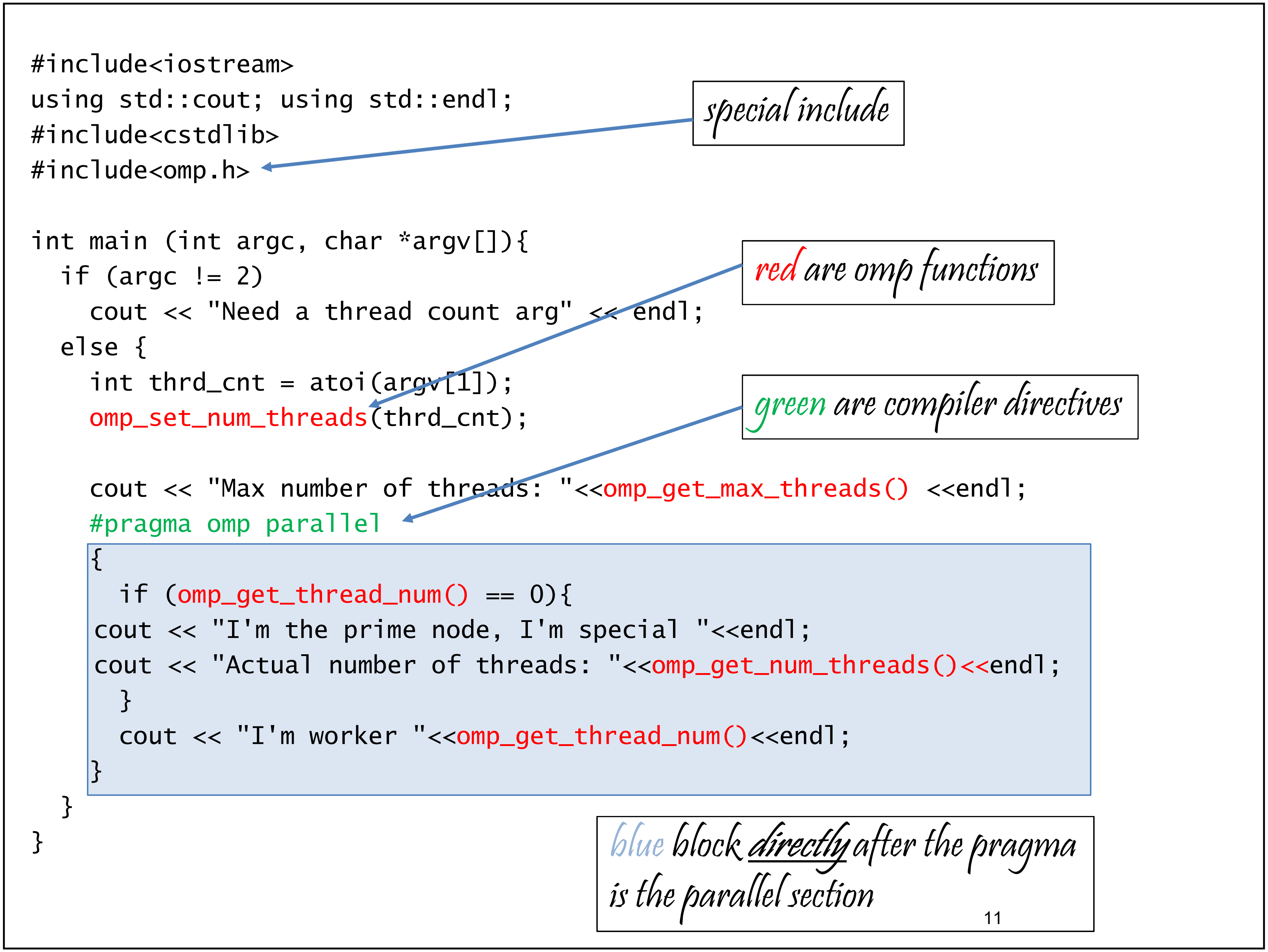
Which is better

- Shared - all processors share a global pool of memory
  - simpler to program
  - bus contention leads to poor scalability
- Distributed - each processor physically has it's own (private) memory associated with it
  - scales well
  - memory management is more difficult

- OpenMP program starts single threaded
- To create additional threads, user starts a parallel region
  - more threads are launched to create a team
  - original (master) thread is part of the team
  - threads “go away” at the end of the parallel region: usually sleep or spin
- Repeat parallel regions as necessary
  - **Fork-join model**



- start with one thread
  - called the master thread (poor term)
- indicated parallel region
  - "team" starts threads to run region code
- region finishes, wait for all threads to get there and join (just like threads). A implicit barrier
- back to the single master
- so on and so forth



```
#include<iostream>
using std::cout; using std::endl;
#include<cstdlib>
#include<omp.h>

int main (int argc, char *argv[]){
  if (argc != 2)
    cout << "Need a thread count arg" << endl;
  else {
    int thrd_cnt = atoi(argv[1]);
    omp_set_num_threads(thrd_cnt);

    cout << "Max number of threads: "<<omp_get_max_threads() <<endl;
    #pragma omp parallel
    {
      if (omp_get_thread_num() == 0){
    cout << "I'm the prime node, I'm special "<<endl;
    cout << "Actual number of threads: "<<omp_get_num_threads()<<endl;
      }
      cout << "I'm worker "<<omp_get_thread_num()<<endl;
    }
  }
}
```

special include

red are omp functions

green are compiler directives

blue block <u>directly</u> after the pragma is the parallel section

compiling and output

```
g++ -fopenmp hello-omp.cpp
./a.out 4
Max number of threads: 4
I'm the prime node, I'm special
Actual number of threads: 4
I'm worker 0
I'm worker 1
I'm worker 2
I'm worker 3
```

Is it really threaded?

- If you get an error in the pragma, not always clear
- by default, it just runs serial
- good to check.
  - for example, not doing `-fopenmp` is an example. No errors and no threading

Some warnings

- no order on I/O. Threads can output in any order
  - if you need order, can be done by concurrency controls
  - could output to multiple files by thread
- OpenMP is free to cache data in a thread and output to memory when it sees fit.
  - `flush` pragma directive

Parallel section run by all

- That parallel section is run in its entirety by each thread (including master)
- Spawn the number of threads you set by the function
  - multiple ways to do this
  - `#pragma omp parallel num_threads(cnt)`
- All threads wait at the end of the section then the master picks up again, an implicit barrier

How many threads did I get?

- OpenMP is not required to give you the number of threads you requested.
  - if you want to know what you received, you have to note it somehow.

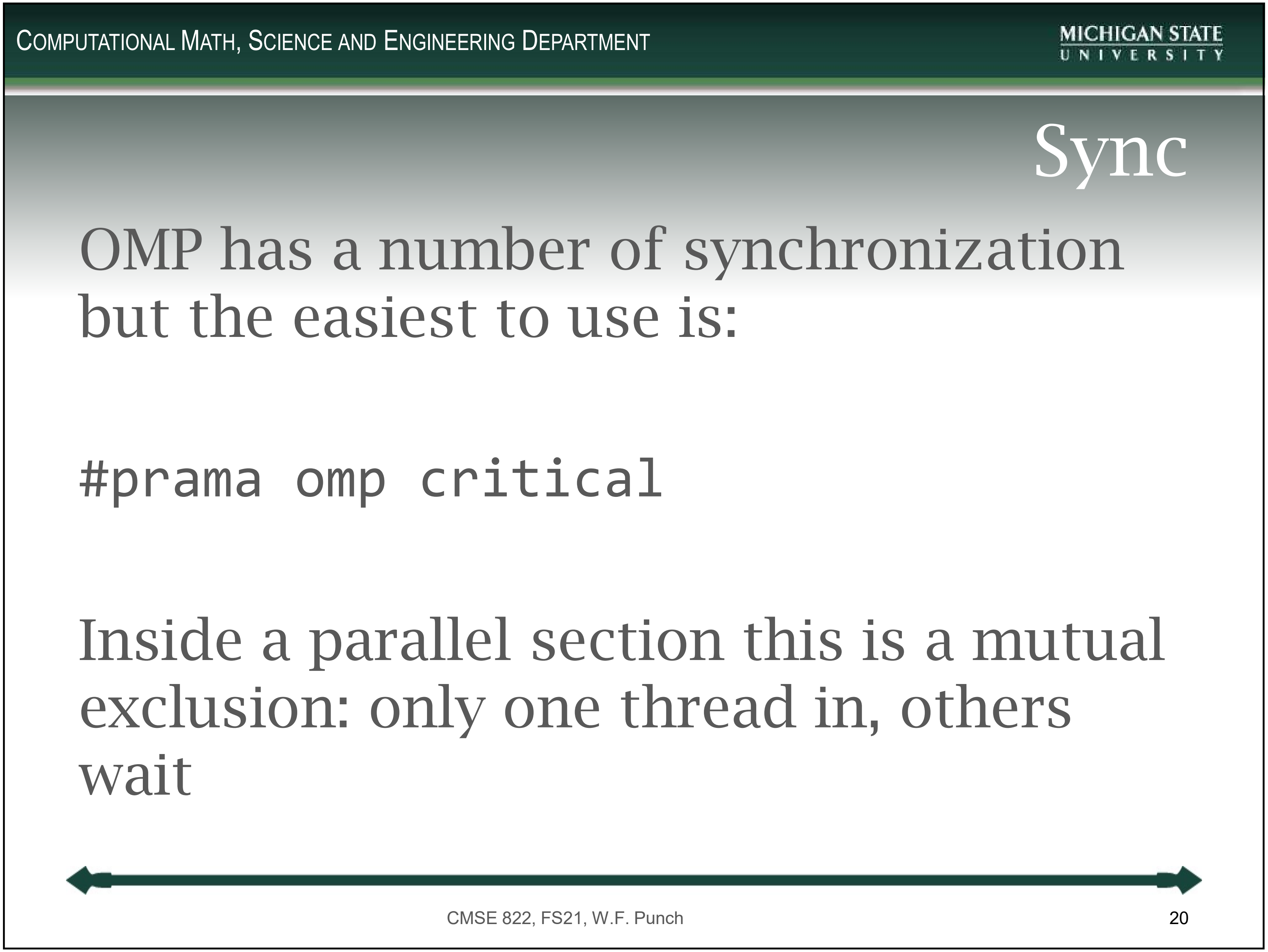
OpenMP scope and sharing

thread local

- global sharing: any variable outside of the parallel block is global to each thread spawned.
  - shared access, no sync (unless we do it)
- thread local: any variables declared inside the parallel block will be local to each thread
  - no sharing of values
  - lost when the thread ends.

pragma control

- `private (list).` List of thread local variables
- `shared (list).` List of shared variables.
- `default(shared | none)`
  - everything shared
  - unknown status and programmer must specify.



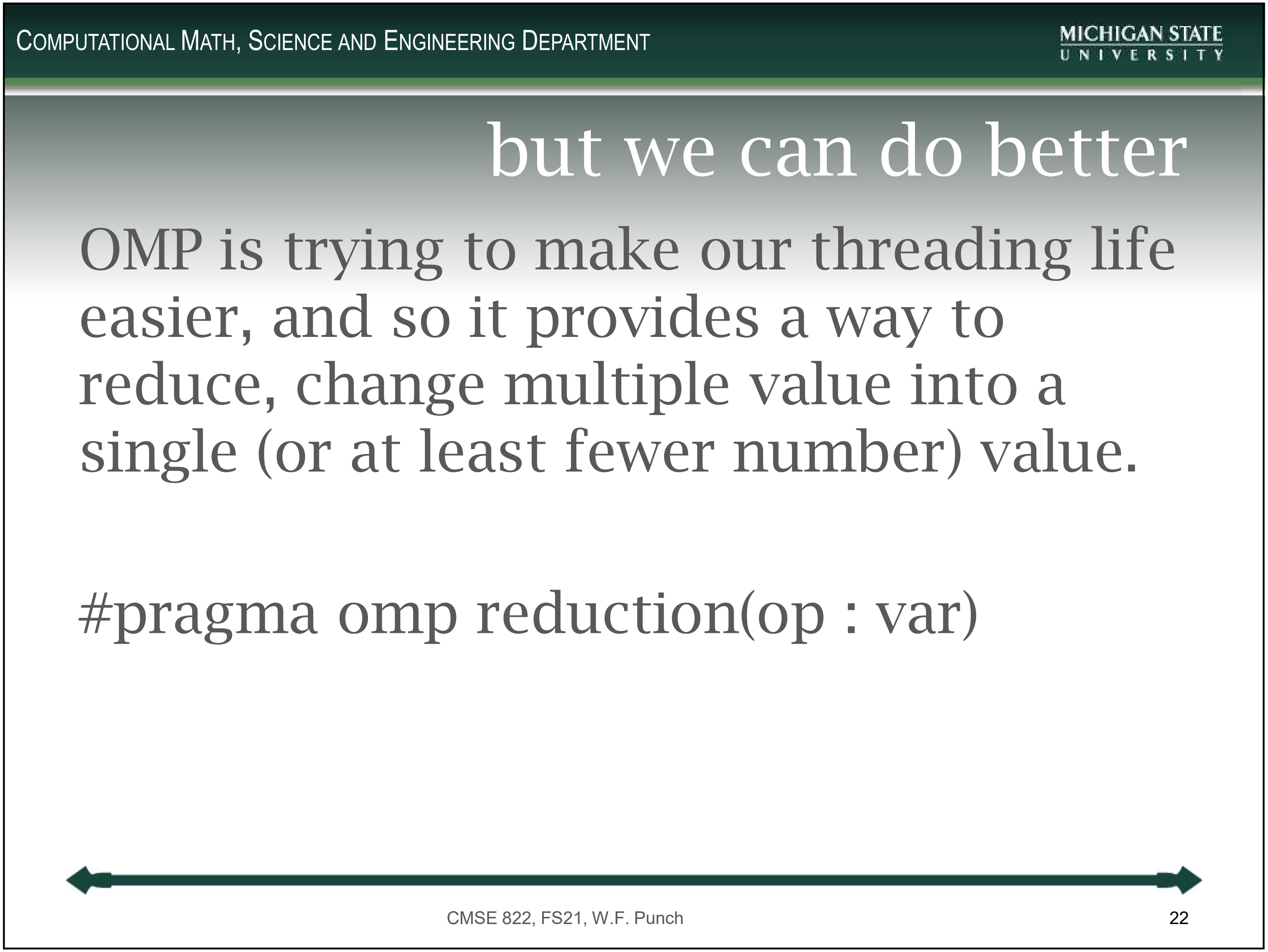
Sync

OMP has a number of synchronization but the easiest to use is:

```
#prama omp critical
```

Inside a parallel section this is a mutual exclusion: only one thread in, others wait

trap1



but we can do better

OMP is trying to make our threading life easier, and so it provides a way to reduce, change multiple value into a single (or at least fewer number) value.

#pragma omp reduction(op : var)

reductions operations

op can be: +, *, -, &, |, ^, &&, ||

Effectively, OpenMP makes a private variable for each thread and then, and then the resulting private variables are "op" together, yielding a result

trap-reduction

OpenMP parallel Region Directive

#pragma omp parallel [*clause list*]

Typical clauses in [*clause list*]

- Conditional parallelization
  - if (scalar expression)
    - Determine whether the parallel construct creates threads
- Degree of concurrency
  - num_threads (integer expresson)
    - number of threads to create
- Date Scoping
  - private (variable list)
    - Specifies variables local to each thread
  - firstprivate (variable list)
    - Similar to the private
    - Private variables are initialized to variable value before the parallel directive
  - shared (variable list)
    - Specifies variables that are shared among all the threads
  - default (data scoping specifier)
    - Default data scoping specifier may be shared or none

more about default

- default(none) : With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.
- default(shared) : assume everything is shared, up to you to declare private.

```
#pragma omp parallel if (is_parallel == 1)
num_threads(8) shared (var_b)  private (var_a)
firstprivate (var_c) default (none)
{
/* structured block */
}
```

- if (is_parallel == 1) num_threads(8)
  - If the value of the variable is_parallel is one, create 8 threads
- shared (var_b)
  - Each thread shares a single copy of variable b
- private (var_a)   firstprivate (var_c)
  - Each thread gets private copies of variable var_a and var_c
  - Each private copy of var_c is initialized with the value of var_c in main thread when the parallel directive is encountered
- default (none)
  - Default state of a variable is specified as none (rather than shared)
  - Signals error if not all variables are specified as shared or private

Nested Sections

For nested sections, the default is that the outer section is parallel and the inner sections are single threaded.